Example: working with matrices

To illustrate some more complicated list manipulation, we will define the basic mathematical functions on matrices. For simplicity, we will just use integer matrices.

As in GAP, a vector will be a list of numbers, and a matrix will be a list of vectors. That is, a (row) vector will be an object of type `[Int]`, and a matrix will be of type `[[Int]]`. But being mathematicians, we want to think about vectors and matrices, not lists of lists. So we will use Haskell's capability to define our own types, either our own names for built-in types, or our own more complicated types. The syntax to do this is:

```
type Vector = [Int]
type Matrix = [[Int]]
```

Anything that we define to have type `Vector` will automatically have all the properties of the type `[Int]`.

We will be working a lot with lists, so we will import the `List` library. Among other conveniences, it contains a built-in transpose function,

```
transpose :: [[a]] -> [[a]]
```

To read about all of the functions in the `List` library, follow the Haskell 98 Report link on our links page, and click on List, item 17 of Part II (check out the handy `nub` function).

We will certainly need to refer to the number of rows and number of columns of a matrix, so we define these functions:

```
numRows ::  Matrix -> Int
numRows = length

numColumns ::  Matrix -> Int
numColumns = length .  head
```

We will be needing zero vectors, so we define a function that will create them:

```
zeroVector ::  Int -> Vector
zeroVector n = replicate n 0
```

Here, we used the Prelude function `replicate :: Int -> a -> [a]`. Given a non-negative integer n and a value `v`, it returns a list of list of length n each of whose terms equals `v`.

We can define scalar products using list comprehension:

```
vectorScalarProduct ::  Int -> Vector -> Vector
vectorScalarProduct n vec = [ n * x | x <- vec ]

matrixScalarProduct ::  Int -> Matrix -> Matrix
matrixScalarProduct n m = [ vectorScalarProduct n row | row <- m ]
```

Our definition of vector addition will (indirectly) use the `zip` function from the Prelude.

```
zip :: [a] -> [b] -> [(a,b)]
```

`zip` is defined by `zip` $[x_1, x_2, \ldots, x_n]$ $[y_1, y_2, \ldots, y_n] = [(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)]$. If one of the lengths is longer than the other one, the extra elements are discarded. Often the reason that you want to zip is to apply a function to each pair. Haskell provides an abbreviation for this:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y :  (zipWith f xs ys)
```

Using `zipWith`, we can quickly define addition:

```
vectorSum ::  Vector -> Vector -> Vector
vectorSum = zipWith (+)

matrixSum ::  Matrix -> Matrix -> Matrix
matrixSum = zipWith vectorSum
```

Using the Prelude function `sum`, which sums the elements of a list, we have a quick definition of dot product:

```
dotProduct :: Vector -> Vector -> Int
dotProduct v w = sum ( zipWith (*) v w )
```

Besides the `sum` function, there is a `prod` function, which takes the product of the elements of a list. And there are the `and` and `or` functions, which take the logical conjunctions and disjunctions of a list of booleans (remember, the logical "and" and "or" operations are `&&` and `||`).

For the matrix product, dot the rows with the columns:

```
matrixProduct :: Matrix -> Matrix -> Matrix
matrixProduct m n = [ map (dotProduct row) (transpose n) | row <- m ]
```

Let's analyze the expression `[ map (dotProduct row) (transpose n) | row <- m ]`. It will be a list calculated as follows:

1. Draw a row `r` from `m`.

2. Evaluate `map (dotProduct r) (transpose n)`: First, notice that `dotProduct r` is a function of type `Vector -> Int`. So `map (dotProduct r) (transpose n)` is of type `[Int]`. Form `transpose n`, that is, a list of row vectors which are the columns of `n`, and apply dot product with `r` to each of those rows, obtaining an integer. This turns the list of vectors `transpose n` into a single vector whose j^{th} entry is the dot product of the row `r` with the j^{th} column of `n`.

3. Add this vector to the list `[ map (dotProduct r) (transpose n) | r <- m ]`.

Now, we will calculate the determinant, using expansion of cofactors. Of course, this is not an efficient algorithm for large matrices, we are just using it as an illustration of how closely the syntax of Haskell follows the mathematical definition of determinant.

To form the adjoint matrix, we will need to be able to remove a row or column. We write a utility function `cut` that removes the entry at a given position from a list. We use 1 for the first position, and so on (this might be a bad idea, since the first element of a Haskell list has index 0). We will use the handy Prelude functions `take` and `drop`:

```
cut :: [a] -> Int -> [a]
cut [ ] n = [ ]
cut xs n
  | n < 1 || n > (length xs) = xs
  | otherwise = (take (n-1) xs) ++ drop n xs
```

Now we use `cut` to remove the i^{th} row and the j^{th} entry of each column:

```
remove :: Matrix -> Int -> Int -> Matrix
remove m i j
  | m == [ ] || i < 1 || i > numRows m || j < 1 || j > numColumns m
     = error "remove:  (i,j) out of range"
  | otherwise = transpose ( cut (transpose ( cut m i ) ) j )
```

The `error` function is a built-in Haskell function that prints out the character string given as its argument and halts execution. Assuming that (i, j) defines a location in the matrix, we calculate the resulting matrix as follows:

1. `cut m i` is `m` with the i^{th} row removed.
2. Take the transpose of `cut m i`, and remove its j^{th} row. These are the entries of the j^{th} column of the untransposed matrix.
3. Transpose back.

Now, to calculate the determinant, we expand along the first row.

```
determinant :: Matrix -> Int
determinant [ ] = error "determinant:  0-by-0 matrix"
determinant [[n]] = n
determinant m
  = sum [ (-1)^ (j+1) * (head m)!!(j-1) * determinant (remove m 1 j) |
  j <- [1..(numColumns m) ] ]
```

An empty matrix gives an error and a 1×1-matrix is handled as a special case. The notation `!!` is Haskell's rather strange notation for list entry, so `aList!!0` is the first element of the list `aList`, and `aList!!7` is its eighth element. In our case `(head m)!!(j-1)` is the j^{th} entry of the first row of `m`. The list comprehension given as `determinant m` expands along the first row, multiplying the j^{th}-entry of the first row by its cofactor, and summing the results.

Finally, we will compute inverses. First, we need the matrix of cofactors.

```
cofactor ::  Matrix -> Int -> Int -> Int
cofactor m i j = (-1)^ (i+j) * determinant (remove m i j)

cofactorMatrix ::  Matrix -> Matrix
cofactorMatrix m =
  [ [ (cofactor m i j) | j <- [1..n] ] | i <- [1..n] ]
  where
  n = length m
```

The last step is to divide by the determinant and take the transpose:

```
inverse ::  Matrix -> Matrix
inverse m = transpose [ [ quot x (determinant m) | x <- row ] |
  row <- (cofactorMatrix m) ]
```

Now, what about testing our routines? Error checking is not as big a problem in Haskell as in most languages, but of course it is still important. As an example, we will write an additional method, called `test`, that will test whether the `inverse` routine is working correctly. The idea is: Represent an elementary matrix with a value v in the (i, j)-place as a triple (i, j, v). A list of such triples represents a list of elementary matrices, and we can take their matrix product. We will calculate its inverse using `inverse`, and test whether it is correct by multiplying the two matrices and comparing to the identity.

```
elemMatrix :: Int -> Int -> Int -> Int -> Matrix
-- elemMatrix n i j v is the n-by-n elementary matrix
-- with v in the (i,j) place
elemMatrix n i j v
  = [ [ entry row column | column <- [1..n] ] | row <- [1..n] ]
  where
  entry x y
    | x == y           = 1
    | x == i && y == j = v
    | otherwise       = 0

idMatrix :: Int -> Matrix
idMatrix n = elemMatrix n 1 1 1

eProduct :: Int -> [(Int,Int,Int)] -> Matrix
-- eProduct n [(Int,Int,Int)] is the product of the elementary matrices
eProduct n [ ] = idMatrix n
eProduct n ((i,j,value):rest) = matrixProduct ( elemMatrix n i j value)
  (eProduct n rest)

minSize :: [(Int,Int,Int)] -> Int
```

```
-- smallest size of matrix for which all elementary matrices are defined
minSize list = maximum (concat [ [i,j] | (i,j,value) <- list ] )

checkInverse :: [(Int,Int,Int)] -> String
checkInverse list =
  "\n M = " ++ (show m) ++ "\nInverse(M) = " ++ (show (inverse m)) ++
  if matrixProduct m (inverse m) == idMatrix n then "\nOK.\n" else "\nError.\n"
  where
  m = eProduct n list
  n = minSize list

list1 :: [(Int,Int,Int)]
list1 = [(1,2,1), (1,3,-1), (1,2,1), (3,2,-2), (3,1,-3)]

list2 :: [(Int,Int,Int)]
list2 = [(1,2,4), (4,2,-1), (4,1,-2), (4,1,1), (1,3,-3), (2,3,2),
  (1,2,2), (1,4,-3), (1,3,-1), (3,2,-1), (3,1,-1)]

test :: IO()
test = putStr (( checkInverse list1 ) ++ ( checkInverse list2 ) )
```

This is our first experience with input-output in Haskell. A Haskell program is isolated from the outside world, and is only allowed to communicate through the IO *monad,* which is denoted by `IO()`. Monads in Haskell are a difficult concept, and they have many different purposes. The IO monad serves as a device that restricts communication in such as way as to maintain the functional purity of Haskell. It allows just enough "imperative" structure to accomplish input-output effectively. In this case, we want to print information to the monitor. We first define a function that creates. Then, we define an object `test`, of type `IO()`, that uses the *put string* function `putstr`. When you enter `test` at the command, its evaluation results in evaluation of the `putstr` function, and the string appears on your monitor.