99 questions/Solutions/93

From HaskellWiki

< 99 questions | Solutions

(***) An arithmetic puzzle

Given a list of integer numbers, find a correct way of inserting arithmetic signs (operators) such that the result is a correct equation. Example: With the list of numbers [2,3,5,7,11] we can form the equations 2-3+5+7 = 11 or 2 = (3*5+7)/11 (and ten others!).

Division should be interpreted as operating on rationals, and division by zero should be avoided.

```
module P93 where

import Control.Monad
import Data.List
import Data.Maybe

type Equation = (Expr, Expr)
data Expr = Const Integer | Binary Expr Op Expr
        deriving (Eq, Show)
data Op = Plus | Minus | Multiply | Divide
        deriving (Bounded, Eq, Enum, Show)
type Value = Rational

-- top-level function: all correct equations generated from the list of
-- numbers, as pretty strings.
puzzle :: [Integer] -> [String]
puzzle ns = map (flip showsEquation "") (equations ns)

-- generate all correct equations from the list of numbers
equations :: [Integer] -> [Equation]
equations [] = error "empty list of numbers"
equations [n] = error "only one number"
equations ns = [(e1, e2) |
                (ns1, ns2) <- splits ns,
                (e1, v1) <- exprs ns1,
                (e2, v2) <- exprs ns2,
                v1 == v2]

-- generate all expressions from the numbers, except those containing
-- a division by zero, or redundant right-associativity.
exprs :: [Integer] -> [(Expr, Value)]
exprs [n] = [(Const n, fromInteger n)]
exprs ns = [(Binary e1 op e2, v) | (ns1, ns2) <- splits ns,
                (e1, v1) <- exprs ns1,
                (e2, v2) <- exprs ns2,
                op <- [minBound..maxBound],
                not (right_associative op e2),
```

```haskell
                v <- maybeToList (apply op v1 v2)]

-- splittings of a list into two non-empty lists
splits :: [a] -> [([a],[a])]
splits xs = tail (init (zip (inits xs) (tails xs)))

-- applying an operator to arguments may fail (division by zero)
apply :: Op -> Value -> Value -> Maybe Value
apply Plus x y = Just (x + y)
apply Minus x y = Just (x - y)
apply Multiply x y = Just (x * y)
apply Divide x 0 = Nothing
apply Divide x y = Just (x / y)

-- e1 op (e2 op' e3) == (e1 op e2) op' e3
right_associative :: Op -> Expr -> Bool
right_associative Plus (Binary _ Plus _) = True
right_associative Plus (Binary _ Minus _) = True
right_associative Multiply (Binary _ Multiply _) = True
right_associative Multiply (Binary _ Divide _) = True
right_associative _ _ = False

-- Printing of equations and expressions

showsEquation :: Equation -> ShowS
showsEquation (l, r) = showsExprPrec 0 l . showString " = " . showsExprPrec 0 r

-- all operations are left associative
showsExprPrec :: Int -> Expr -> ShowS
showsExprPrec _ (Const n) = shows n
showsExprPrec p (Binary e1 op e2) = showParen (p > op_prec) $
        showsExprPrec op_prec e1 . showString (opName op) .
                showsExprPrec (op_prec+1) e2
  where op_prec = precedence op

precedence :: Op -> Int
precedence Plus = 6
precedence Minus = 6
precedence Multiply = 7
precedence Divide = 7

opName :: Op -> String
opName Plus = "+"
opName Minus = "-"
opName Multiply = "*"
opName Divide = "/"
```

Unlike the Prolog solution, I've eliminated solutions like `"1+(2+3) = 6"` as a trivial variant of `"1+2+3 = 6"` (cf the function `right_associative`). Apart from that, the Prolog solution is shorter because it uses built-in evaluation and printing of expressions.