99 questions/Solutions/46

From HaskellWiki

< 99 questions | Solutions

(**) Define predicates and/2, or/2, nand/2, nor/2, xor/2, impl/2 and equ/2 (for logical equivalence) which succeed or fail according to the result of their respective operations; e.g. and(A,B) will succeed, if and only if both A and B succeed.

A logical expression in two variables can then be written as in the following example: and(or(A,B),nand(A,B)).

Now, write a predicate table/3 which prints the truth table of a given logical expression in two variables.

The first step in this problem is to define the Boolean predicates:

```
-- NOT negates a single Boolean argument
not' :: Bool -> Bool
not' True  = False
not' False = True

-- Type signature for remaining logic functions
and',or',nor',nand',xor',impl',equ' :: Bool -> Bool -> Bool

-- AND is True if both a and b are True
and' True True = True
and' _    _    = False

-- OR is True if a or b or both are True
or' False False = False
or' _     _     = True

-- NOR is the negation of 'or'
nor'  a b = not' $ or'  a b

-- NAND is the negation of 'and'
nand' a b = not' $ and' a b

-- XOR is True if either a or b is True, but not if both are True
xor' True  False = True
xor' False True  = True
xor' _     _     = False

-- IMPL is True if a implies b, equivalent to (not a) or (b)
impl' a b = (not' a) `or'` b

-- EQU is True if a and b are equal
equ' True  True  = True
equ' False False = True
```

```
equ' _     _     = False
```

The above implementations build each logic function from scratch; they could be shortened using Haskell's builtin equivalents:

```
and'  a b = a && b
or'   a b = a || b
nand' a b = not (and' a b)
nor'  a b = not (or' a b)
xor'  a b = not (equ' a b)
impl' a b = or' (not a) b
equ'  a b = a == b
```

Some could be reduced even further using Pointfree style:

```
and' = (&&)
or'  = (||)
equ' = (==)
```

The only remaining task is to generate the truth table; most of the complexity here comes from the string conversion and IO. The approach used here accepts a Boolean function `(Bool -> Bool -> Bool)`, then calls that function with all four combinations of two Boolean values, and converts the resulting values into a list of space-separated strings. Finally, the strings are printed out by mapping
putStrLn
across the list of strings:

```
table :: (Bool -> Bool -> Bool) -> IO ()
table f = mapM_ putStrLn [show a ++ " " ++ show b ++ " " ++ show (f a b)
                                | a <- [True, False], b <- [True, False]]
```

The table function in Lisp supposedly uses Lisp's symbol handling to substitute variables on the fly in the expression. I chose passing a binary function instead because parsing an expression would be more verbose in haskell than it is in Lisp. Template Haskell could also be used :)

The table function can be generalized to work for any given binary function and domain.

```
table :: (Bool -> Bool -> Bool) -> String
table f = printBinary f [True, False]

printBinary :: (Show a, Show b) => (a -> a -> b) -> [a] -> String
printBinary f domain = concatMap (++ "\n") [printBinaryInstance f x y | x <- domain, y <- dom

printBinaryInstance :: (Show a, Show b) => (a -> a -> b) -> a -> a -> String
printBinaryInstance f x y = show x ++ " " ++ show y ++ " " ++ show (f x y)
```


Category:

- Programming exercise spoilers

- Recent content is available under a simple permissive license.