

Logic Haskell Exercises

Young W. Lim

2018-09-13 Thr

- 1 Based on
- 2 Logic
 - Using TAMO.hs

"The Haskell Road to Logic, Maths, and Programming", K. Doets and J. V. Eijck

I, the copyright holder of this work, hereby publish it under the following licenses: GNU head Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled GNU Free Documentation License.

CC BY SA This file is licensed under the Creative Commons Attribution ShareAlike 3.0 Unported License. In short: you are free to share and make derivative works of the file under the conditions that you appropriately attribute it, and that you distribute it only under a license compatible with this one.

Using TAMO.hs

```
module TAMO
```

```
where
```

```
:load TAMO
```

Quantifiers as Functions

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> :load TAMO
[1 of 1] Compiling TAMO                ( TAMO.hs, interpreted )
Ok, modules loaded: TAMO.
*TAMO> any (>3) [0 ..]
True
*TAMO> any (<3) [0 ..]
True
*TAMO> any (<-1) [0 ..]

<interactive>:5:6:
  parse error on input '<-'
  Perhaps this statement should be within a 'do' block?
*TAMO> any (< -1) [0 ..]

^CInterrupted. (takes too long)
```

- Datatype Bool

```
data Bool = False | True
```

- Negation (not)

```
not :: Bool -> Bool
not True = False
not False = True
```

- Conjunction (&&)

```
(&&) :: Bool -> Bool -> Bool
False && x = False
True && x = x
```

- Disjunction (||)

```
(||) :: Bool -> Bool -> Bool
False || x = x
True || x = True
```

- Implication (\implies)

```
(==>) :: Bool -> Bool -> Bool~  
x ==> y = (not x) || y~
```

- Equivalence (\iff)

```
(<=>) :: Bool -> Bool -> Bool  
x <=> y = x == y
```

- Exclusive or (∇)

```
(<+>) :: Bool -> Bool -> Bool  
x <+> y = x /= y
```

Fixity Declaration in Haskell

- `infix` non-associativity
- `infixl` left-associativity
- `infixr`
right-associativity
- specifies a precedence level from 0 to 9
 - 0 (weakest)
 - 9 (strongest)
 - 10 (normal application)

http://zvon.org/other/haskell/Outputsyntax/fixityQdeclaration_reference.html

Fixity Declaration Example

```
main = print (1 +++ 2 *** 3)
```

```
infixr 6 +++  
infixr 7 ***,///
```

```
(+++)  
a +++ b = a + 2*b
```

```
(***)  
a *** b = a - 4*b
```

```
(///)  
a /// b = 2*a - 3*b
```

$(1 \text{ +++ } (2 \text{ *** } 3)) = (1 \text{ +++ } (2 - 4*3)) = (1 \text{ +++ } -10) = 1 + 2*(-10) = -19$

http://zvon.org/other/haskell/Outputsyntax/fixityQdeclaration_reference.html

Fixity of Connectives Definition

- `infix 1 ==>`
- `infix 1 <=>`
- `infixr 2 <+>`

Connectives Examples

```
*Main> True ==> True      *Main> True <=> True      *Main> True <+> True
True
*Main> True ==> False     *Main> True <=> False     *Main> True <+> False
False
*Main> False ==> True     *Main> False <=> True    *Main> False <+> True
True
*Main> False ==> False   *Main> False <=> False   *Main> False <+> False
True
-----
*Main> (==>) True True    *Main> (<=>) True True    *Main> (<+>) True True
True
*Main> (==>) True False  *Main> (<=>) True False  *Main> (<+>) True False
False
*Main> (==>) False True  *Main> (<=>) False True  *Main> (<+>) False True
True
*Main> (==>) False False *Main> (<=>) False False *Main> (<+>) False False
True
```

Evaluating Connectives

P=True, Q=False

```
~ P ^ ((P -> Q) <-> ~(Q ^ ~ P))
: T :   T : F   : : F : : T
F   :     F   : :   : F
    :         : :   F
    :         : : T
    :         F
    F
```

```
~ P ^ ((P -> Q) <-> ~(Q ^ ~ P))
F T F   T F F   F T F F F T
```

No Multiple Declarations in Haskell

- in ghci, can change the value of a binding

```
P=True  
Q=False
```

```
P=False  
Q=True
```

- in ghc, cannot change the value of a binding

```
Error  
Multiple declarations of 'P'  
Multiple declarations of 'Q'
```

Evaluating Connectives in Haskell

```
p = True
q = False
```

```
formula1 = (not p) && (p ==> q) <=> not (q && (not p))
           F      T      T      F      F      T
           F      F      F      F      T
           F
```

```
formula2 p q = ((not p) && (p ==> q) <=> not (q && (not p)))
```

```
-----
*Main> formula1
False
*Main> formula2 True True
False
*Main> formula2 True False
False
*Main> formula2 False True
False
*Main> formula2 False False
True

*Main> :t formula1
formula1 :: Bool

Bool type value

*Main> :t formula2
formula2 :: Bool -> Bool -> Bool

a function value
taking two Bool type values
returning a Bool type value
```

Validity Check

- True for every possible combination of propositional arguments
- a **formula** is valid iff it is true under every interpretation
- an argument is valid iff it is impossible for the premises to be true and for the conclusion to be false

List Comprehension

- to enumerate all the possible cases of a truth table a list comprehension can be used

- 2 proposition truth table

```
[(p, q) | p <- [True, False], q <- [True, False]]
```

```
[(True, True), (True, False), (False, True), (False, False)]
```

- 3 proposition truth table

```
[(p, q, r) | p <- [True, False], q <- [True, False], r <- [True, False]]
```

```
[(True, True, True), (True, True, False), (True, False, True),  
 (True, False, False), (False, True, True), (False, True, False),  
 (False, False, True), (False, False, False)]
```


Boolean Function Types

- Boolean functions with only one variable
`Bool -> Bool`
- Boolean functions with two variables
`Bool -> Bool -> Bool`
- Boolean functions with three variables
`Bool -> Bool -> Bool -> Bool`

- Boolean functions `bf` with only one variable

```
valid1 :: (Bool -> Bool) -> Bool
valid1 bf = (bf True) && (bf False)
```

- Boolean functions `bf` with two variables

```
valid2 :: (Bool -> Bool -> Bool) -> Bool
valid2 bf = (bf True True)
           && (bf True False)
           && (bf False True)
           && (bf False False)
```

Validity Check Functions using list comprehensions

- Boolean functions bf with two variables

```
valid2 :: (Bool -> Bool -> Bool) -> Bool
valid2 bf = and [ bf p q | p <- [True,False],
                    q <- [True,False]]
```

- Boolean functions bf with three variables

```
valid3 :: (Bool -> Bool -> Bool -> Bool) -> Bool
valid3 bf = and [ bf p q r | p <- [True,False],
                          q <- [True,False],
                          r <- [True,False]]
```

- Boolean functions bf with only one variable

```
valid4 :: (Bool -> Bool -> Bool -> Bool -> Bool) -> Bool
valid4 bf = and [ bf p q r s | p <- [True,False],
                              q <- [True,False],
                              r <- [True,False],
                              s <- [True,False]]
```

Validity Function Examples

- a valid function with only one variable

$P \vee \neg P$: tautology

```
excluded_middle :: Bool -> Bool
excluded_middle p = p || not p
```

- a valid function with two variables

$P \Rightarrow (Q \Rightarrow P)$: tautology

$(P \Rightarrow Q) \Rightarrow P$: not valid $(P \Rightarrow Q) \supset P$

```
form1 p q = p ==> (q ==> p)
form2 p q = (p ==> q) ==> p
```

```
*Main> valid1 excluded_middle
True
*Main> valid2 form1
True
*Main> valid2 form2
False
```

Logical Equivalence Check Function Types

- check logical equivalence for propositional functions with 1 parameter

```
logEquiv1 :: (Bool -> Bool) -> -- bf1 function type
            (Bool -> Bool) -> -- bf2 function type
            Bool
```

- check logical equivalence for propositional functions with 2 parameters

```
logEquiv2 :: (Bool -> Bool -> Bool) -> -- bf1 function type
            (Bool -> Bool -> Bool) -> -- bf2 function type
            Bool
```

- check logical equivalence for propositional functions with 2 parameters

```
logEquiv3 :: (Bool -> Bool -> Bool -> Bool) -> -- bf1 function type
            (Bool -> Bool -> Bool -> Bool) -> -- bf2 function type
            Bool
```

Logical Equivalence Check Function Definitions

- check logical equivalence for propositional functions with 1 parameter

```
logEquiv1 bf1 bf2 = (bf1 True ==> bf2 True) &&
                    (bf1 False ==> bf2 False)
```
- check logical equivalence for propositional functions with 2 parameters

```
logEquiv2 bf1 bf2 = and [(bf1 p q) ==> (bf2 p q) | p <- [True,False],
                          q <- [True,False]]
```
- check logical equivalence for propositional functions with 3 parameters

```
logEquiv3 bf1 bf2 = and [(bf1 p q r) ==> (bf2 p q r) | p <- [True,False],
                          q <- [True,False],
                          r <- [True,False]]
```

Logical Equivalence Examples

- $q \oplus q = F$
- $p \oplus F = p$
- $(p \oplus q) \oplus q = p \oplus (q \oplus q) = p \oplus F = p$

formula3 p q = p

formula4 p q = (p <+> q) <+> q

formula5 p q = p <=> ((p <+> q) <+> q)

```
-----  
*Main> logEquiv2 formula3 formula4 -- logical equivalent  
True  
*Main> valid2 formula5 -- tautology  
True
```