```
{-

 LET True  = \ x y. x
 LET False = \ x y. y y

 LET AND = \ a b. a b (\z. False)

 False AND True = False
 AND False True = (\ a b. a b (\z. False)) False True
                = (\ b. False b (\z. False)) True
                = False True (\z. False)
                = (\ x y. y y) True (\z .False)
                = (\ y. y y) (\z. False)
                = (\z. False) (\z. False)
                = False
  AND True q = (\ a b. a b (\z. False)) True q
             = (\ b. True b (\z. False)) q
             = True q (\z. False)
             = (\ x y. x) q (\z. False)
             = (\ y. q) (\z. False)
             = q

-}

{-

 LET True  = \ x y. x x
 LET False = \ x y. y

 LET OR = \ a b. a (\z. True) b

 True OR False = True
 OR True False = (\ a b. a (\z. True) b) True False
               = (\ b. True (\z. True) b) False
               = True (\z. True) False
               = (\ x y. x x) (\z. True) False
               = (\ y. (\z.True) (\z.True)) False
               = (\z.True) (\z.True)
               = True
 OR False q = (\ a b. a (\z. True) b) False q
            = (\ b. False (\z. True) b) q
            = False (\z. True) q
            = (\ x y. y) (\z. True) q
            = (\ y. y) q
            = q

-}


--------------------------------------------------------------------
--------------------------------------------------------------------

data TExpr a
  = Var a
  | Val Double
  | Add (TExpr a) (TExpr a)
  | Sub (TExpr a) (TExpr a)
  | Mul (TExpr a) (TExpr a)
  | Div (TExpr a) (TExpr a)
  | Pow (TExpr a) (TExpr a)
  deriving (Show, Eq)


toPow :: Eq a => TExpr a -> TExpr a
toPow v@(Var _) = v
toPow v@(Val _) = v
toPow (Add l r) = Add (toPow l) (toPow r)
toPow (Sub l r) = Sub (toPow l) (toPow r)
toPow (Div l r) = Div (toPow l) (toPow r)
toPow (Pow l r) = Pow (toPow l) (toPow r)
toPow (Mul l r) =
  let
    (fl,num,v) = resolveMul l r
  in
    if fl then Pow (Var v) (Val num)
    else Mul (toPow l) (toPow r)

resolveMul :: Eq a => TExpr a -> TExpr a -> (Bool,Double,a)
resolveMul (Var v1) (Var v2)
  | v1==v2 = (True,2.0,v1)
  | True = (False,0.0,undefined)
resolveMul (Mul l r) (Var v) =
  let
    (fl,num,w) = resolveMul l r
  in
    (fl && w==v,num+1.0,w)
resolveMul (Var v) (Mul l r) =
  let
    (fl,num,w) = resolveMul l r
  in
    (fl && w==v,num+1.0,w)
resolveMul (Mul l1 r1) (Mul l2 r2) =
  let
    (f1,n1,w1) = resolveMul l1 r1
    (f2,n2,w2) = resolveMul l2 r2
  in
    (f1 && f2 && w1==w2,n1+n2,w1)
resolveMul _ _ = (False,0.0,undefined)
```

```haskell
sumConst :: TExpr a -> TExpr a
sumConst v@(Var _) = v
sumConst v@(Val _) = v
sumConst (Sub l r) = Sub (sumConst l) (sumConst r)
sumConst (Mul l r) = Add (sumConst l) (sumConst r)
sumConst (Div l r) = Div (sumConst l) (sumConst r)
sumConst (Pow l r) = Pow (sumConst l) (sumConst r)
sumConst (Add l r) =
  let
    (fl,v) = mkSum l r
  in
    if fl then (Val v)
    else Add (sumConst l) (sumConst r)

mkSum :: TExpr a -> TExpr a -> (Bool,Double)
mkSum (Val v1) (Val v2) = (True,v1+v2)
mkSum (Add l r) (Val v) =
  let
    (fl,w) = mkSum l r
  in
    (fl,w+v)
mkSum (Val v) (Add l r) =
  let
    (fl,w) = mkSum l r
  in
    (fl,w+v)
mkSum (Mul l1 r1) (Mul l2 r2) =
  let
    (fl1,w1) = mkSum l1 r1
    (fl2,w2) = mkSum l2 r2
  in
    (fl1 && fl2, w1+w2)
mkSum _ _ = (False,0.0)



-----------------------------------------------------------------------
-----------------------------------------------------------------------


llen :: [t] -> Int
llen [] = 0                    -- 1
llen (_:xs) = 1 + llen xs      -- 2


inc2 :: a -> Int -> Int
inc2 _ n = 1+n                 -- 3

flen :: [t] -> Int
flen = foldr inc2 0            -- 4

{-

foldr _ v [] = v                           -- 5
foldr f v (x:xs) = f x (foldr f v xs)      -- 6

---

llen xs = flen xs

--
xs==[]

L = llen xs =
  llen [] =|1
  0

P = flen xs =
  flen [] =|4
  foldr inc2 0 [] =|5
  0

L==P

--
I.P.
llen as = flen as

xs=(a:as)

L = llen xs =
  llen (a:as) =| 2
  1 + llen as =|I.P.
  1 + flen as

P = flen xs =
  flen (a:as) =|4
  foldr inc2 0 (a:as) =|6
  inc2 a (foldr inc2 0 as) =|3
  1 + (foldr inc2 0 as) =|4
  1 + (flen as)

L==P

-}
```

```haskell
anyT :: [Bool] -> Bool
anyT [] = False                -- 1
anyT (x:xs) = x || anyT xs     -- 2

fanyT :: [Bool] -> Bool
fanyT = foldr (||) False       -- 3

{-

foldr _ v [] = v                          -- 4
foldr f v (x:xs) = f x (foldr f v xs)     -- 5

---

anyT xs == fanyT xs

--
xs==[]

L = anyT xs =
  anyT [] =|1
  False

P = fanyT xs =
  fanyT [] =|3
  foldr (||) False [] =|4
  False

L==P

--
I.P.
anyT as = fanyT as

xs=(a:as)

L = anyT xs =
  anyT (a:as) =|2
  a || anyT as =|I.P.
  a || fanyT as

P = fanyT xs =
  fanyT (a:as) =|3
  foldr (||) False (a:as) =|5
  a || (foldr (||) False as) =|3
  a || fanyT aa

L==P

-}


-- EOF
```