# Monad Exam Solutions

## 1 2021 Q2 (csexams)

#### 1.0.1 Assumptions

- These will not be needed in the exam. These are the assumptions of the functions we are given, from looking at its type signature and relating it back to our tutorials.

```
ins :: String -> Int -> Dict -> Dict
ins s i d = (s,i) : d

lkp :: (Monad m, MonadFail m)  => String -> Dict -> m Int
lkp s [] = fail $ "lkp - Couldn't find: " ++ show s
lkp s ((s1,i):ds)
    | s == s1   = return i
    | otherwise = lkp s ds
```

### 1.1 Part (a)

- Third case of eval you divide by two numbers, eval d e2 could be zero and you would get a divide by zero error run time error.

### 1.2 Part (b)

1. Mul from data Expr does not have a pattern in eval function, and will fail a pattern match.
2. A pattern match where you do not specify the correct arguments with the correct types for the data type Expr, e.g. eval [("a",1)] (Let "a" "b" "c"). Let expects String Expr Expr. And so a pattern patch error will occur.

### 1.3 Part (c)

```
eval :: (Monad m, MonadFail m)  => Dict -> Expr -> m Int
eval _ (K i) = return i
eval d (V s) = do
    case lkp s d of
        Nothing -> fail ("prop-variable is undefined: "++s)
        Just i -> return i
eval d (Dvd e1 e2) = do
    evale1 <- eval d e1 -- laziness makes it efficient
    evale2 <- eval d e2
    if evale2 == 0
        then fail "divide by zero"
        else return (evale1 `div` evale2) -- evale1 will now evaluate here
eval d (Mul e1 e2) = do
    evale1 <- eval d e1
    evale2 <- eval d e2
    return (evale1 * evale2)
eval d (Let v e1 e2) = do
    evale1 <- eval d e1
    eval (ins v evale1 d) e2
```

## 2 2021 Q2

#### 2.0.1 Assumptions

- These will not be needed in the exam. These are the assumptions of the functions we are given, from looking at its type signature and relating it back to our tutorials.

```
ins :: String -> Bool -> Dict -> Dict
ins s b d = (s,b) : d

lkp :: (Monad m, MonadFail m)  => String -> Dict -> m Bool
lkp s [] = fail $ "lkp - Couldn't find: " ++ show s
lkp s ((s1,b):ds)
    | s == s1   = return i
    | otherwise = lkp s ds
```

### 2.1 Part (a)

- Fourth case of eval you divide by two numbers, 3 `div` 0, you will get a divide by zero error run time error.

### 2.2 Part (b)

1. Not from data Prop does not have a pattern in beval function, and will fail a pattern match.
2. Fourth case pattern matches beval _ (B True) but does not pattern match with beval _ (B False) and will cause a pattern match error.

### 2.3 Part (c)

```
eval :: (Monad m, MonadFail m)  => Dict -> Prop -> m Bool
beval _ (B b) = return b
beval d (P s) = do
    case lkp s d of
        Nothing -> fail $ "prop-variable is undefined: " ++ s
        Just b  -> return b
beval d (Not p1) = do
    rp1 <- beval d p1
    return (not rp1)
beval d (And p1 p2) = do
    rp1 <- beval d p1
    rp2 <- beval d p2
    return (rp1 && rp2)
beval d (Let v p1 p2) = do
    b <- beval d p1
    beval (ins v b d) p2
```

## 3 2019 Q2

#### 3.0.1 Notes

- If you want to test this code in ghci on your computer you will need use deriving Show.

```
data BinTree = BTwo BinTree Int String BinTree
             | BOne Int String
             | BZero
             deriving Show
```

### 3.1 Part (a)

1. A pattern match error will occur when matchinf for 'BZero' will cause a pattern match error.
2. A pattern match error will occur for when z < i in lookup (BTwo left i s right) z.
3. A pattern match error will occur when calling lookup recursively in the second case. e.g. lookup z right attempts to pattern match lookup with lookup :: Int -> BinTree -> String however that function does not exist. You must swap the parameters for it to work.

### 3.2 Part (b)

- Implementation using guards

```
lookup :: BinTree -> Int -> Maybe String
lookup BZero _ = Nothing
lookup (BOne i s) z
  | z == i    = Just s
  | otherwise = Nothing
lookup (BTwo left i s right) z
  | z == i    = Just s
  | z > i     = lookup right z
  | otherwise = lookup left z
```

- Implementation using if, else, do notation

```
lookup :: BinTree -> Int -> Maybe String
lookup BZero _ = Nothing
lookup (BOne i s) z = do
  if z == i
     then Just s
     else Nothing
lookup (BTwo left i s right) z = do
  if z == i
     then Just s
     else if z > i
             then lookup right z
             else lookup left z
```

### 3.3 Part (c)

```
insert :: Int -> String -> BinTree -> BinTree
insert key val BZero = BOne key val
insert key val (BOne k v)
   | key == k = BOne key val
   | key < k  = BTwo (BOne key val) k v BZero
   | key > k  = BTwo BZero k v (BOne key val)
insert key val (BTwo left k v right)
   | key == k = BTwo left key val right
   | key < k  = BTwo (insert key val left) k v right
   | key > k  = BTwo left k v (insert key val right)
```

# 4 2018 Q2

### 4.0.1 Notes

- If you want to test this code in ghci on your computer you will need use deriving Show.

```
data BinTree = BNil
             | BOne Int String
             | BTwo BinTree Int String BinTree
             deriving Show
```

## 4.1 Part (a)

1. A pattern match error will occur when matchinf for 'BNil' will cause a pattern match error.
2. A pattern match error will occur for when x == i in lookup (BTwo left i s right) z.
3. A pattern match error will occur because the first case does not pattern match for when z /= i.

## 4.2 Part (b)

- Implementation using guards

```
lookup :: BinTree -> Int -> Maybe String
lookup BNil _ = Nothing
lookup (BOne i s) z
  | z == i    = Just s
  | z /= i = Nothing
lookup (BTwo left i s right) z
  | z == i    = Just s
  | z > i     = lookup right z
  | otherwise = lookup left z
```

- Implementation using if, else, do notation

```
lookup :: BinTree -> Int -> Maybe String
lookup BNil _ = Nothing
lookup (BOne i s) z = do
  if z == i
     then Just s
     else Nothing
lookup (BTwo left i s right) z = do
  if z == i
     then Just s
     else if z > i
             then lookup right z
             else lookup left z
```

### 4.3 Part (c)

- For generic error handling using monads we will need to use part b with our do notation.

```
lookup :: (Monad m, MonadFail m) => BinTree -> Int -> m String
lookup BNil z = fail $ "Could not find val: " ++ show z
lookup (BOne i s) z = do
  if z == i
     then return s
     else lookup BNil z
lookup (BTwo left i s right) z = do
  if z == i
     then return s
     else if z > i
             then lookup' right z
             else lookup' left z
```

## 5 2018 Q2

#### 5.0.1 Notes

- If you want to test this code in ghci on your computer you will need use deriving Show.

```
data Tree = Empty
          | Single Int String
          | Many Tree Int String Tree
          deriving Show
```

### 5.1 Part (a)

1. A pattern match error will occur when matching for 'Empty' will cause a pattern match error.
2. A pattern match error will occur because the first case does not pattern match for when x $/=$ i.

### 5.2 Part (b)

- Implementation using guards

```
search :: Int -> Tree -> Maybe String
search _ Empty = Nothing
search x (Single i s)
  | x == i = Just s
  | x /= i = Nothing
search x (Many left i s right)
  | x == i = Just s
  | x > i  = search x right
  | x < i  = search x left
```

- Implementation using if, else, do notation

```
search :: Int -> Tree -> Maybe String
search _ Empty = Nothing
search x (Single i s) = do
  if x == i
     then Just s
     else Nothing
search x (Many left i s right) = do
  if x == i
     then Just s
     else if x > i
             then search x right
             else search x left
```

### 5.3 Part (c)

- For generic error handling using monads we will need to use part b with our do notation.

```
search :: (Monad m, MonadFail m) => Int -> Tree -> m String
search x Empty = fail $ "Could not find val: " ++ show x
search x (Single i s) = do
  if x == i
     then return s
     else search x Empty
search x (Many left i s right) = do
  if x == i
     then return s
     else if z > i
             then search x right
             else search x left
```

# 6 2017 Q2

## 6.1 Part (c)

1. First runtime error occurs with first case of 'Empty' where it calls undefined which is a function that produces a runtime error
2. Second runtime error occurs with second case of search where it fails to check x /= i.
3. Third runtime error occurs with third case of search where it fails to check for x > i.
4. Fourth runtime error occurs with third case of search where it fails to check for x == i.

# 7 2017 Q3

### 7.0.1 Assumptions

- These will not be needed in the exam. These are the assumptions of the functions we are given, from looking at its type signature and relating it back to our tutorials.

```
lkp :: String -> Dict -> Maybe Int
lkp s [] = Nothing
lkp s ((s1,i):ds)
    | s == s1   = Just i
    | otherwise = lkp s ds

ins :: (String, Int) -> Dict -> Dict
ins toDict@(s,i) d = toDict : d
```

### 7.0.2 Notes

- If you want to test this code in ghci on your computer you will need use deriving Show.

```
data Expr = K Int
          | V String
          | Add Expr Expr
          | Dvd Expr Expr
          | Where Expr String Expr
          deriving Show
```

### 7.1 Part (a)

```
eval :: Dict -> Expr -> Maybe Int
eval _ (K i) = Just i
eval d (V s) = lkp s d
eval d (Add e1 e2) = do
  evale1 <- eval d e1
  evale2 <- eval d e2
  Just (evale1 + evale2)
eval d (Dvd e1 e2) = do
  evale1 <- eval d e1
  evale2 <- eval d e2
  if evale2 == 0
     then Nothing
     else Just (evale1 `div` evale2) -- lazy evaluation on evale1
eval d (Where e1 v e2) = do
  i <- eval d e2
  eval (ins (v,i) d) e1
```

## 8 2016 Q2

### 8.1 Part (c)

1. A pattern match will result in a runtime error for when 'Empty' Tree is not pattern matched.
2. A pattern match will result in a runtime error for the second case of search with x $/=$ i.
3. A pattern match will result in a runtime error for the second case of search with x $>$ i.
4. A pattern match will result in a runtime error for the second case of search with x $<$ i.
5. A pattern match will result in a runtime error for the first case of search with search x right, search takes in a Tree and then an Int however in this case we have it backwards.

# 9 2016 Q3

### 9.0.1 Assumptions

- These will not be needed in the exam. These are the assumptions of the functions we are given, from looking at its type signature and relating it back to our tutorials.

```
lkp :: String -> Dict -> Maybe Int
lkp s [] = Nothing
lkp s ((s1,i):ds)
    | s == s1   = Just i
    | otherwise = lkp s ds

ins :: String -> Int -> Dict -> Dict
ins s i d = (s,i) : d
```

### 9.0.2 Notes

- If you want to test this code in ghci on your computer you will need use deriving Show.

```
data Expr = K Int
          | V String
          | Add Expr Expr
          | Dvd Expr Expr
          | Let Expr String Expr
          deriving Show
```

## 9.1 Part (a)

```
eval :: Dict -> Expr -> Maybe Int
eval _ (K i) = Just i
eval d (V s) = lkp s d
eval d (Add e1 e2) = do
  evale1 <- eval d e1
  evale2 <- eval d e2
  Just (evale1 + evale2)
eval d (Dvd e1 e2) = do
  evale1 <- eval d e1
  evale2 <- eval d e2
  if evale2 == 0
     then Nothing
     else Just (evale1 `div` evale2) -- lazy evaluation on evale1
eval d (Let e1 v e2) = do
  i <- eval d e2
  eval (ins v i d) e1
```

# 10 2014 Q3

### 10.0.1 Notes

- If you want to test this code in ghci on your computer you will need use deriving Show.

```
data Tree = Empty
          | Single Int String
          | Many Tree Int String Tree
          deriving Show
```

## 10.1 Part (a)

1. A pattern match error will result in a runtime error for when search Empty is run
2. A pattern match error will result in a runtime error for when in the second case of search where x $/=$ i.
3. A pattern match error will result in a runtime error for when in the second case of search where x $<$ i.

## 10.2 Part (b)

- Same exact answer as second 2018 Q2 (b)

## 10.3 Part (c)

- For generic error handling using monads we will need to use part b with our do notation.

```
search :: Int -> Tree -> Either String String
search x Empty = Left $ "Could not find val: " ++ show x
search x (Single i s) = do
  if x == i
     then Right s
     else search x Empty
search x (Many left i s right) = do
  if x == i
     then Right s
     else if z > i
             then search x right
             else search x left
```

# 11 2013 Q3

### 11.0.1 Assumptions

- These will not be needed in the exam. These are the assumptions of the functions we are given, from looking at its type signature and relating it back to our tutorials.

```
lkp :: String -> Dict -> Maybe Int
lkp s [] = Nothing
lkp s ((s1,i):ds)
    | s == s1   = Just i
    | otherwise = lkp s ds

ins :: String -> Int -> Dict -> Dict
ins s i d = (s,i) : d
```

## 11.1 Part (a)

- A runtime error can occur if in the 4th case of eval you evaluate the eval d e2 to 0, in turn dividing something by zero. That would give a divide by zero runtime error.

## 11.2 Part (b)

```
eval :: Dict -> Expr -> Maybe Int
eval _ (K i) = Just i
eval d (V s) = lkp s d
eval d (Add e1 e2) = do
  evale1 <- eval d e1
  evale2 <- eval d e2
  Just (evale1 + evale2)
eval d (Dvd e1 e2) =
  evale1 <- eval d e1
  evale2 <- eval d e2
  if evale2 == 0
     then Nothing
     else Just (evale1 `div` evale2)
eval d (Let v e1 e2) = do
  i <- eval d e1
  eval (ins v i d) e2
```

### 11.3 Part (c)

```
eval :: Dict -> Expr -> Either String Int
eval _ (K i) = Right i
eval d (V s) = do
  case lkp s d of
    Nothing -> Left $ "v-variable is undefined: " ++ s
    Just b  -> Right b
eval d (Add e1 e2) = do
  evale1 <- eval d e1
  evale2 <- eval d e2
  Right (evale1 + evale2)
eval d (Dvd e1 e2) =
  evale1 <- eval d e1
  evale2 <- eval d e2
  if evale2 == 0
     then Left "Divide by zero"
     else Right (evale1 `div` evale2)
eval d (Let v e1 e2) = do
  i <- eval d e1
  eval (ins v i d) e2
```

# 12 2012 Q2

## 12.1 Part (b)

```
data Expr = Val Float
          | Var String
          | Add Expr Expr
          | Divide Expr Expr
          | Def String Expr Expr

type Dictionary k d = [(k,d)]

lkp :: Eq k => Dictionary k d -> k -> Maybe d
lkp [] _ = Nothing
lkp ( (k,v) : ds ) name
  | name == k = Just v
  | otherwise = lkp ds name

set :: Dictionary k d -> k -> d -> Dictionary k d
set d k v = (k,v) : d

eval :: Dictionary String Float -> Expr -> Maybe Float
eval _ (Val x)   = Just x
eval d (Var i)   = lkp d i
eval d (Add x y) = do
  evalx <- eval d x
  evaly <- eval d y
  Just (evalx + evaly)
eval d (Divide x y) = do
  evalx <- eval d x
  evaly <- eval d y
  if evaly == 0.0
     then Nothing
     else Just $ evalx / evaly
eval d (Def s x y) = do
  v <- eval d x
  eval (set d s v) y
```