Model Examen

```
type Name = String

data Term = Var Name | Con Integer | Term :+: Term | Lam Name Term | App Term Term | Out Term
deriving (Show)

-- Definirea unui interpretator folosind monada Writer

newtype Writer a = Writer {runWriter :: (a, String)}

instance Monad Writer where
  return a = Writer(a, "")
  ma >>= k = let (a, log1) = runWriter ma
                 (b, log2) = runWriter (k a)
             in Writer (b, log1 ++ log2)

instance Applicative Writer where
  pure = return
  mf <*> ma = do {f <- mf; a <-ma; return (f a)}

instance Functor Writer where
  fmap f ma = pure f <*> ma

showM :: Show a => Writer a -> String
showM ma = "Output: " ++ w ++ "\nValue: " ++ show a
           where (a, w) = runWriter ma

data Value = Num Integer | Fun (Value -> Writer Value) | Wrong
type Environment = [(Name, Value)]

instance Show Value where
  show (Num x) = show x
  show (Fun _) = "<function>"
  show Wrong = "<wrong>"

interp :: Term -> Environment -> Writer Value
interp (Var x) env = get x env
interp (Con i) _ = return $ Num i
interp (t1 :+: t2) env = do
                           v1 <- interp t1 env
                           v2 <- interp t2 env
                           add v1 v2
interp (Lam x e) env = return $ Fun (\v-> interp e ((x,v):env))
interp (App t1 t2) env = do
  f <- interp t1 env
  v <- interp t2 env
  apply f v
interp (Out t) env = do
  v<-interp t env
  tell (show v ++ "; ")
  return v

tell :: String -> Writer ()
tell s = Writer ((), s)

apply :: Value -> Value -> Writer Value
apply (Fun k) v = k v
apply _ _ = return Wrong

get :: Name -> Environment -> Writer Value
get x env = case [v | (y,v)<-env, x==y] of
              (v:_) -> return v
              _ -> return Wrong

add :: Value -> Value -> Writer Value
```

```
add (Num i) (Num j) = return (Num $ i+j)
add _ _ = return Wrong

test :: Term -> String
test t = showM (interp t [])

pgm :: Term
pgm = App (Lam "x" ((Var "x") :+: (Var "x"))) ((Out (Con 10)) :+: (Out (Con 11)))

pgmW4::Term
pgmW4 = App (Var "y") (Lam "y" (Out (Con 3)))
```

EXERCITIUL 1:

```
-- Definirea unui interpretator folosind monada ce combina Writer cu Maybe;
-- in cazul aparitiei unei erori se va intoarce rezultatul Nothing fara a afisa outputul
cumulat pana in acel moment.

type Name = String

data Term = Var Name | Con Integer | Term :+: Term | Lam Name Term | App Term Term | Out Term
deriving (Show)

newtype MaybeWriter a = MW {getValue :: Maybe(a, String)}

instance Monad MaybeWriter where
  return x = MW (Just(x, ""))
  ma >>= f = case a of
               Nothing -> MW Nothing
               Just (x, w) -> case getValue (f x) of
                                Nothing -> MW Nothing
                                Just (y, v) -> MW (Just(y, w++v))
               where a = getValue ma


instance Applicative MaybeWriter where
  pure = return
  mf <*> ma = do {f <- mf; a <-ma; return (f a)}

instance Functor MaybeWriter where
  fmap f ma = pure f <*> ma

showM :: Show a => MaybeWriter a -> String
showM ma =
           case a of
             Nothing -> "Nothing"
             Just (x, w) -> "Output: " ++ w ++ "\nValue: " ++ show x
           where a = getValue ma

data Value = Num Integer | Fun (Value -> MaybeWriter Value)
type Environment = [(Name, Value)]

instance Show Value where
  show (Num x) = show x
  show (Fun _) = "<function>"

interp :: Term -> Environment -> MaybeWriter Value
interp (Var x) env = get x env
interp (Con i) _ = return $ Num i
interp (t1 :+: t2) env = do
                           v1 <- interp t1 env
                           v2 <- interp t2 env
                           add v1 v2
interp (Lam x e) env = return $ Fun (\v-> interp e ((x,v):env))
interp (App t1 t2) env = do
  f <- interp t1 env
  v <- interp t2 env
```

```
  apply f v
interp (Out t) env = do
  v<-interp t env
  tell (show v ++ "; ")
  return v


tell :: String -> MaybeWriter ()
tell s = MW (Just ((), s))

apply :: Value -> Value -> MaybeWriter Value
apply (Fun k) v = k v
apply _ _ = MW Nothing

get :: Name -> Environment -> MaybeWriter Value
get x env = case [v | (y,v)<-env, x==y] of
              (v:_) -> return v
              _ -> MW Nothing

add :: Value -> Value -> MaybeWriter Value
add (Num i) (Num j) = return (Num $ i+j)
add _ _ = MW Nothing

test :: Term -> String
test t = showM (interp t [])

pgm :: Term
pgm = App (Lam "x" ((Var "x") :+: (Var "x"))) ((Out (Con 10)) :+: (Out (Con 11)))

pgmW4::Term
pgmW4 = App (Var "y") (Lam "y" (Out (Con 3)))
```

Exercitiul 2:

```
-- Adaugam :/: la Term ce reprezinta impartirea; consideram eroare impartirea la 0

type Name = String

data Term = Var Name | Con Integer | Term :+: Term | Term :/: Term | Lam Name Term | App Term
Term | Out Term deriving (Show)

newtype MaybeWriter a = MW {getValue :: Maybe(a, String)}

instance Monad MaybeWriter where
  return x = MW (Just(x, ""))
  ma >>= f = case a of
               Nothing -> MW Nothing
               Just (x, w) -> case getValue (f x) of
                                Nothing -> MW Nothing
                                Just (y, v) -> MW (Just(y, w++v))
               where a = getValue ma

instance Applicative MaybeWriter where
  pure = return
  mf <*> ma = do {f <- mf; a <-ma; return (f a)}

instance Functor MaybeWriter where
  fmap f ma = pure f <*> ma

showM :: Show a => MaybeWriter a -> String
showM ma =
           case a of
             Nothing -> "Nothing"
             Just (x, w) -> "Output: " ++ w ++ "\nValue: " ++ show x
           where a = getValue ma

data Value = Num Integer | Fun (Value -> MaybeWriter Value)
```

```
type Environment = [(Name, Value)]

instance Show Value where
  show (Num x) = show x
  show (Fun _) = "<function>"

interp :: Term -> Environment -> MaybeWriter Value
interp (Var x) env = get x env
interp (Con i) _ = return $ Num i
interp (t1 :+: t2) env = do
                           v1 <- interp t1 env
                           v2 <- interp t2 env
                           add v1 v2
interp (t1 :/: t2) env = do
                           v1 <- interp t1 env
                           v2 <- interp t2 env
                           divM v1 v2
interp (Lam x e) env = return $ Fun (\v-> interp e ((x,v):env))
interp (App t1 t2) env = do
  f <- interp t1 env
  v <- interp t2 env
  apply f v
interp (Out t) env = do
  v<-interp t env
  tell (show v ++ "; ")
  return v

tell :: String -> MaybeWriter ()
tell s = MW (Just ((), s))

apply :: Value -> Value -> MaybeWriter Value
apply (Fun k) v = k v
apply _ _ = MW Nothing

get :: Name -> Environment -> MaybeWriter Value
get x env = case [v | (y,v)<-env, x==y] of
              (v:_) -> return v
              _ -> MW Nothing

add :: Value -> Value -> MaybeWriter Value
add (Num i) (Num j) = return (Num $ i+j)
add _ _ = MW Nothing

divM :: Value -> Value -> MaybeWriter Value
divM (Num i) (Num 0) = MW Nothing
divM (Num i) (Num j) = return (Num $ i `div` j)
divM _ _ = MW Nothing

test :: Term -> String
test t = showM (interp t [])

pgm :: Term
pgm = App (Lam "x" ((Var "x") :+: (Var "x"))) ((Out (Con 10)) :/: (Out (Con 2)))

pgmW2::Term
pgmW2 = App (Lam "x" ((Var "x") :+: (Var "x"))) ((Out (Con 10)) :/: (Out (Con 0)))
```

EXERCITUL 3: Foldr universalitate

```
insertAtRec :: Int -> [Int] -> Int -> [Int]
insertAtRec val [] _ = [val]
insertAtRec val (x:xs) poz
                          |poz <= 1 = val:(x:xs)
                          |otherwise = x:(insertAtRec val xs (poz-1))
```

```
insertAtFold :: Int -> [Int] -> Int -> [Int]
insertAtFold val lista poz = (foldr op unit lista) poz val
                             where
                             unit _ x = [x]
                             (a `op` r) poz val
                                            |poz <= 1 = val:(a:(r 1 val))
                                            |otherwise = a:(r (poz-1) val)
```