Prosty program:

Hello world

```
main :: IO ()
main = putStrLn "Hello, World!"
```

suma:

```
main :: IO ()
main = getLine >>= \x -> getLine >>= \y -> putStrLn (show (read x + read y))
```

lub

```
main :: IO ()
main = getLine >>= \x ->
        getLine >>= \y ->
        return(show(read x + read y)) >>= putStrLn
```

dziel

```
main::IO()
main = getLine >>= \x→
        getLine>>= \y →
        return(show(read x `div` read y) >>= putStrLn
```

```
data Maybe a = Just a | Nothing deriving Show
```

```
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

```
instance Applicative Maybe where
 pure                = Just
 Just f <*> Just a = Just (f a)
 _          <*> _          = Nothing

instance Monad Może where
 return                          = pure
 Just a >>= f              = f a
 Nothing       >>= f              = Nothing
```

```
instance Applicative Maybe where
   pure = Just
   Nothing <*> _ = Nothing
   (Just f) <*> something = fmap f something

instance Monad Maybe where
```

```
    return x = Just x
    Nothing >>= f = Nothing
    Just x>>= f = f x
    fail _ = Nothing
```

Drzrwo

```
data MyTree a = MyLeaf a
                | MyNode (MyTree a) (MyTree a)
                deriving (Show)

instance Functor MyTree where
    fmap f (MyLeaf x) = MyLeaf (f x)
    fmap f (MyNode x y) = MyNode (fmap f x) (fmap f y)

instance Applicative MyTree where
    pure = MyLeaf
    (MyLeaf f) <*> (MyLeaf x) = MyLeaf (f x)
    (MyLeaf f) <*> (MyNode x y) = MyNode (fmap f x) (fmap f y)

instance Monad MyTree where
    return = MyLeaf
    (MyLeaf x) >>= f = f x
    (MyNode x y) >>= f = MyNode (x >>= f) (y >>= f)
```

Lub

```
data MyTree a = MyEmptyNode
                | MyFilledNode a (MyTree a) (MyTree a)

instance Functor MyTree where
    fmap f MyEmptyNode          = MyEmptyNode
    fmap f (MyFilledNode x y z) = MyFilledNode (f x) (fmap f y) (fmap f z)

instance Applicative MyTree where
    pure x = MyFilledNode x MyEmptyNode MyEmptyNode
    (MyFilledNode f fy fz) <*> (MyFilledNode x y z) = MyFilledNode (f x) MyEmptyNode
MyEmptyNode
    _ <*> _ = MyEmptyNode


instance Monad MyTree where
    return x = MyFilledNode x MyEmptyNode MyEmptyNode
    (MyFilledNode x y z) >>= f = f x
    MyEmptyNode >>= _ = MyEmptyNode
```

Lista

```
instance Functor [] where
  fmap = map
```

instance Applicative [] where
pure x = [x]
fs <*> xs= [f x | f ← fs, x ← xs]

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs= [f x | f ← fs, x ← xs]

instance Monad [] where
    return x = [x]
    xs >>= f= concat(map f xs)
    fail _ = []
```

Either

```
instance Functor (Either e) where
  fmap f (Right x) = Right (f x)
  fmap _ l         = l

instance Applicative (Either e) where
  pure          = Right
  (Right f) <*> v = fmap f v
  l         <*> _ = l

instance Monad (Either e) where
     return = Right
     Right m >>= k = k m
     Left e  >>= _ = Left e


data Either a b = Left a | Right b

instance Functor (Either a) where
   fmap _ (Left x) = Left x
   fmap f (Right y) = Right (f y)

instance Applicative (Either e) where
   pure        = Right
   Left e <*> _ = Left e
   Right f <*> r = fmap f r

instance Monad (Either e) where

   Left l >>= _ = Left l
   Right r >>= k = k r
```

```
main :: IO ()
main = putStrLn "Hello, World!"
```

Dziel.hs

```
main:: IO()

main = getLine >>= \x ->
        getLine >>= \y ->
        return(show(read x `div` read y)) >>= putStrLn
```

Suma.hs

```
main :: IO ()
main = getLine >>= \x ->
    getLine >>= \y ->
    return(show(read x + read y)) >>= putStrLn
```

Może.hs

```
data Może a = Wartość a | Nic deriving Show

instance Functor Może where
  fmap f Nic         = Nic
  fmap f (Wartość w) = Wartość (f w)
```

```
instance Applicative Może where
  pure                    = Wartość
  Wartość f <*> Wartość a = Wartość (f a)
  _         <*> _         = Nic
```

```
instance Monad Może where
  return                 = pure
  Wartość a >>= f        = f a
  Nic       >>= f        = Nic
```

można zastosować do Maybe:

Może=Maybe

Nic=Nothing

Wartość= Just

```
newtype Stan s a = Stan (s -> (a,s))
```

```
instance Functor (Stan s) where
  fmap f (Stan g) = Stan $ \s0 -> let (a, s1) = g s0
                                  in (f a, s1)
```

```
instance Applicative (Stan s) where
  pure a    = Stan $ \s0 -> (a, s0)
  (Stan f) <*> (Stan a) = Stan $ \s0 -> let (f', s1) = f s0
```

```haskell
                                (a', s2) = a s1
                              in (f' a', s2)


instance Monad (Stan s) where
  return = pure


  f >>= k = Stan $ \s0 -> let Stan ff = f
                              (y,s1)  = ff s0
                              Stan gg = k y
                            in gg s1
```

Ewaulator.hs

```haskell
import Control.Monad.Identity
import Control.Monad.Except
import Control.Monad.State


-- Ewaluator wyrażeń arytmetycznych (+, -, *, /)
-- Funkcjonalność:
-- 1.) Obsługa błędów (dzielenie przez zero)
-- 2.) Licznik operacji arytmetucznych wykonanych w trakcie obliczeń
-- 3.) Generowanie logu z obliczeń
```

```
-- A. Typ dla wyrażeń

data Expr = Lit Int | Add Expr Expr | Sub Expr Expr | Mul Expr Expr | Div Expr Expr deriving (Show,Read)

-- B. Typ dla wyniku

type Value = Int

-- C. Zwykły ewaluator funkcyjny

eval :: Expr -> Value
eval (Lit a)   = a
eval (Add a b) = eval a + eval b
eval (Sub a b) = eval a - eval b
eval (Mul a b) = eval a * eval b
eval (Div a b) = eval a `div` eval b

-- D. Konwersja ewaluatora na styl monadowy (w monadzie Identity)

type Eval1 a = Identity a

runEval1 = runIdentity

eval1 :: Expr -> Eval1 Value
eval1 (Lit a)   = return a
eval1 (Add a b) = do v1 <- eval1 a
```

```
                    v2 <- eval1 b
                    return (v1 + v2)
eval1 (Sub a b) = do { v1 <- eval1 a; v2 <- eval1 b; return (v1 - v2) }
eval1 (Mul a b) = do { v1 <- eval1 a; v2 <- eval1 b; return (v1 * v2) }
eval1 (Div a b) = do { v1 <- eval1 a; v2 <- eval1 b; return (v1 `div` v2) }

-- E. Dodanie obsługi błędów

type Eval2 a = ExceptT String Identity a

runEval2 = runIdentity . runExceptT

eval2 :: Expr -> Eval2 Value
eval2 (Lit a)   = return a
eval2 (Add a b) = do v1 <- eval2 a
                    v2 <- eval2 b
                    return (v1 + v2)
eval2 (Sub a b) = do { v1 <- eval2 a; v2 <- eval2 b; return (v1 - v2) }
eval2 (Mul a b) = (*) <$> eval2 a <*> eval2 b
eval2 (Div a b) = do v1 <- eval2 a
                    v2 <- eval2 b
                    if v2 == 0
                      then throwError "Dzielenie przez zero."
                      else return (v1 `div` v2)

-- F. Dodanie licznika
```

```haskell
type Eval3 a = StateT Int (ExceptT String Identity) a

runEval3 e = runIdentity $ runExceptT $ runStateT e 0

eval3 :: Expr -> Eval3 Value
eval3 (Lit a)   = return a
eval3 (Add a b) = modify (+1) >> (+) <$> eval3 a <*> eval3 b
eval3 (Sub a b) = modify (+1) >> (-) <$> eval3 a <*> eval3 b
eval3 (Mul a b) = modify (+1) >> (*) <$> eval3 a <*> eval3 b
eval3 (Div a b) = do modify (+1)
                     v1 <- eval3 a
                     v2 <- eval3 b
                     if v2 == 0
                       then throwError "Dzielenie przez zero."
                       else return (v1 `div` v2)

-- G. Dodanie logu
```

Transformer.hs

```haskell
module Transformers where

import Control.Monad.Trans.Identity
import Control.Monad.Identity
```

```haskell
import Control.Monad.Except
import Control.Monad.Reader
import Control.Monad.State
import Control.Monad.Writer

import Control.Monad.Trans.Class

import Data.Maybe
import qualified Data.Map as Map

import qualified Control.Monad.Fail as Fail

type Name   =  String

data Exp    =  Lit Integer
            |  Var Name
            |  Plus Exp Exp
            |  Abs Name Exp
            |  App Exp Exp
            deriving (Show)

data Value  =  IntVal Integer
            |  FunVal Env Name Exp
            deriving (Show)

type Env    =  Map.Map Name Value
```

```
eval0                 ::  Env -> Exp -> Value
eval0 env (Lit i)       =   IntVal i
eval0 env (Var n)       =   fromJust (Map.lookup n env)
eval0 env (Plus e1 e2)  =   let  IntVal i1  = eval0 env e1
                                 IntVal i2  = eval0 env e2
                            in IntVal (i1 + i2)
eval0 env (Abs n e)     =   FunVal env n e
eval0 env (App e1 e2)   =   let  val1  = eval0 env e1
                                 val2  = eval0 env e2
                            in case val1 of
                                 FunVal env' n body -> eval0 (Map.insert n val2 env') body


exampleExp = Lit 12 `Plus` (App (Abs "x" (Var "x")) (Lit 4 `Plus` Lit 2))


instance Fail.MonadFail Identity where
  fail = Fail.fail


type Eval1 a  =   Identity a


runEval1          ::  Eval1 a -> a
```

```
runEval1 ev       =   runIdentity ev



eval1                    ::  Env -> Exp -> Eval1 Value
eval1 env (Lit i)       =   return $ IntVal i
eval1 env (Var n)       =   return $ fromJust $ Map.lookup n env
eval1 env (Plus e1 e2)  =   do  IntVal i1  <- eval1 env e1
                                IntVal i2  <- eval1 env e2
                                return $ IntVal (i1 + i2)
eval1 env (Abs n e)     =   return $ FunVal env n e
eval1 env (App e1 e2)   =   do  val1  <- eval1 env e1
                                val2  <- eval1 env e2
                                case val1 of
                                  FunVal env' n body -> eval1 (Map.insert n val2 env') body



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


type Eval2 alpha = ExceptT String Identity alpha



runEval2     :: Eval2 alpha -> Either String alpha
runEval2 ev  = runIdentity (runExceptT ev)



eval2a                   ::  Env -> Exp -> Eval2 Value
```

```
eval2a env (Lit i)       =   return $ IntVal i

eval2a env (Var n)       =   return $ fromJust $ Map.lookup n env

eval2a env (Plus e1 e2)  =   do  IntVal i1  <- eval2a env e1

                                 IntVal i2  <- eval2a env e2

                                 return $ IntVal (i1 + i2)

eval2a env (Abs n e)     =   return $ FunVal env n e

eval2a env (App e1 e2)   =   do  val1  <- eval2a env e1

                                 val2  <- eval2a env e2

                                 case val1 of

                                   FunVal env' n body ->

                                     eval2a (Map.insert n val2 env') body
```

```
eval2b                   ::  Env -> Exp -> Eval2 Value

eval2b _   (Lit i)       =   return $ IntVal i

eval2b env (Var n)       =   maybe (throwError ("undefined variable: " ++ n)) return $ Map.lookup n
env

eval2b env (Plus e1 e2)  =   do  e1'  <- eval2b env e1

                                 e2'  <- eval2b env e2

                                 case (e1', e2') of

                                   (IntVal i1, IntVal i2) -> return $ IntVal (i1 + i2)

                                   _                      -> throwError "type error"

eval2b env (Abs n e)     =   return $ FunVal env n e

eval2b env (App e1 e2)   =   do  val1  <- eval2b env e1

                                 val2  <- eval2b env e2

                                 case val1 of
```

```
                  FunVal env' n body -> eval2b (Map.insert n val2 env') body
                  _                  -> throwError "type error"
```

```
eval2c                :: Env -> Exp -> Eval2 Value
eval2c env (Lit i)      = return $ IntVal i
eval2c env (Var n)      = maybe (throwError ("undefined variable: " ++ n)) return $ Map.lookup n
env
eval2c env (Plus e1 e2) = do IntVal i1 <- eval2c env e1
                             IntVal i2 <- eval2c env e2
                             return $ IntVal (i1 + i2)
eval2c env (Abs n e)    = return $ FunVal env n e
eval2c env (App e1 e2)  = do FunVal env' n body <- eval2c env e1
                             val2               <- eval2c env e2
                             eval2c (Map.insert n val2 env') body
```

```
eval2                 :: Env -> Exp -> Eval2 Value
eval2  env (Lit i)      = return $ IntVal i
eval2  env (Var n)      = case Map.lookup n env of
                            Nothing -> throwError ("unbound variable: " ++ n)
                            Just val -> return val
eval2  env (Plus e1 e2) = do e1'  <- eval2  env e1
```

```
                        e2'  <- eval2  env e2
                        case (e1', e2') of
                          (IntVal i1, IntVal i2) -> return $ IntVal (i1 + i2)
                          _                      -> throwError "type error in addition"
eval2  env (Abs n e)    =   return $ FunVal env n e
eval2  env (App e1 e2)  =   do  val1  <- eval2  env e1
                        val2  <- eval2  env e2
                        case val1 of
                          FunVal env' n body -> eval2  (Map.insert n val2 env') body
                          _                  -> throwError "type error in application"
```

```
type Eval3 alpha = ReaderT Env (ExceptT String Identity) alpha

-- instance MonadExcept e m -> MonadExcept e (ReaderT r m) where

runEval3     :: Env -> Eval3 alpha -> Either String alpha
runEval3 env ev  = runIdentity (runExceptT (runReaderT ev env))

eval3            ::  Exp -> Eval3 Value
eval3 (Lit i)      =   return $ IntVal i
eval3 (Var n)      =   do  env <- ask
                     case Map.lookup n env of
```

```
                     Nothing -> throwError ("unbound variable: " ++ n)
                     Just val -> return val
eval3 (Plus e1 e2)  =   do  e1'  <- eval3 e1
                   e2'  <- eval3 e2
                   case (e1', e2') of
                     (IntVal i1, IntVal i2) -> return $ IntVal (i1 + i2)
                     _                      -> throwError "type error in addition"
eval3 (Abs n e)     =   do  env <- ask
                   return $ FunVal env n e
eval3 (App e1 e2)   =   do  val1  <- eval3 e1
                   val2  <- eval3 e2
                   case val1 of
                     FunVal env' n body -> local (const (Map.insert n val2 env')) (eval3 body)
                     _                  -> throwError "type error in application"



--------------------------------------------------------------------------------------------------
type Eval4 alpha = ReaderT Env (ExceptT String (StateT Integer Identity)) alpha


runEval4           ::  Env -> Integer -> Eval4 alpha -> (Either String alpha, Integer)
runEval4 env st ev  =   runIdentity (runStateT (runExceptT (runReaderT ev env)) st)


tick :: (Num s, MonadState s m) => m ()
```

```
tick = do  st <- get
           put (st + 1)



eval4               ::  Exp -> Eval4 Value
eval4 (Lit i)       =   do  tick
                        return $ IntVal i
eval4 (Var n)       =   do  tick
                        env <- ask
                        case Map.lookup n env of
                          Nothing  -> throwError ("unbound variable: " ++ n)
                          Just val -> return val
eval4 (Plus e1 e2)  =   do  tick
                        e1'  <- eval4 e1
                        e2'  <- eval4 e2
                        case (e1', e2') of
                         (IntVal i1, IntVal i2) -> return $ IntVal (i1 + i2)
                         _                      -> throwError "type error in addition"
eval4 (Abs n e)     =   do  tick
                        env <- ask
                        return $ FunVal env n e
eval4 (App e1 e2)   =   do  tick
                        val1  <- eval4 e1
                        val2  <- eval4 e2
                        case val1 of
                          FunVal env' n body -> local (const (Map.insert n val2 env')) (eval4 body)
```

```haskell
                    _                 -> throwError "type error in application"


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

type Eval4' a = ReaderT Env (StateT Integer (ExceptT String Identity)) a

runEval4'           ::  Env -> Integer -> Eval4' alpha -> (Either String (alpha, Integer))
runEval4' env st ev  =   runIdentity (runExceptT (runStateT (runReaderT ev env) st))


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

type Eval5 a = ReaderT Env  (ExceptT String (WriterT [String] (StateT Integer Identity))) a

runEval5            ::  Env -> Integer -> Eval5 alpha -> ((Either String alpha, [String]), Integer)
runEval5 env st ev  =
    runIdentity (runStateT (runWriterT (runExceptT (runReaderT ev env))) st)

eval5               ::  Exp -> Eval5 Value
eval5 (Lit i)       =   do  tick
                            return $ IntVal i
eval5 (Var n)       =   do  tick
                            tell [n]
                            env <- ask
                            case Map.lookup n env of
                              Nothing -> throwError ("unbound variable: " ++ n)
                              Just val -> return val
```

```
eval5 (Plus e1 e2)  =   do  tick
                        e1'  <- eval5 e1
                        e2'  <- eval5 e2
                        case (e1', e2') of
                          (IntVal i1, IntVal i2) ->
                              return $ IntVal (i1 + i2)
                          _ -> throwError "type error in addition"
eval5 (Abs n e)     =   do  tick
                        env <- ask
                        return $ FunVal env n e
eval5 (App e1 e2)   =   do  tick
                        val1  <- eval5 e1
                        val2  <- eval5 e2
                        case val1 of
                           FunVal env' n body ->
                              local (const (Map.insert n val2 env'))
                                (eval5 body)
                           _ -> throwError "type error in application"



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

type Eval6 a = ReaderT Env  (ExceptT String (WriterT [String] (StateT Integer IO))) a



runEval6           ::  Env -> Integer -> Eval6 alpha -> IO ((Either String alpha, [String]), Integer)
runEval6 env st ev  =
```

```
    runStateT (runWriterT (runExceptT (runReaderT ev env))) st



eval6              ::  Exp -> Eval6 Value
eval6 (Lit i)       =   do  tick
                        liftIO $ print i
                        return $ IntVal i
eval6 (Var n)       =   do  tick
                        tell [n]
                        env <- ask
                        case Map.lookup n env of
                          Nothing -> lift $ throwError ("unbound variable: " ++ n)
                          Just val -> return val
eval6 (Plus e1 e2)  =   do  tick
                        e1'  <- eval6 e1
                        e2'  <- eval6 e2
                        case (e1', e2') of
                          (IntVal i1, IntVal i2) ->
                              return $ IntVal (i1 + i2)
                          _ -> lift $ throwError "type error in addition"
eval6 (Abs n e)     =   do  tick
                        env <- ask
                        return $ FunVal env n e
eval6 (App e1 e2)   =   do  tick
                        val1  <- eval6 e1
                        val2  <- eval6 e2
```

```
                      case val1 of
                        FunVal env' n body ->
                          local (const (Map.insert n val2 env'))
                            (eval6 body)
                        _ -> lift $ throwError "type error in application"
```

```
eval4'              ::  Exp -> Eval4' Value
eval4' (Lit i)       =   return $ IntVal i
eval4' (Var n)       =   do env <- ask
                      case Map.lookup n env of
                        Nothing -> throwError ("unbound variable: " ++ n)
                        Just val -> return val
eval4' (Plus e1 e2)  =   do  e1'  <- eval4' e1
                       e2'  <- eval4' e2
                       case (e1', e2') of
                        (IntVal i1, IntVal i2) ->
                           return $ IntVal (i1 + i2)
                        _ -> throwError "type error in addition"
eval4' (Abs n e)     =   do  env <- ask
                       return $ FunVal env n e
```

```
eval4' (App e1 e2)   =   do  val1  <- eval4' e1
                             val2  <- eval4' e2
                             case val1 of
                               FunVal env' n body ->
                                  local (const (Map.insert n val2 env'))
                                    (eval4' body)
                               _ -> throwError "type error in application"



main = do let r0 = eval0 Map.empty exampleExp
          print r0
          let r1 = runEval1 (eval1 Map.empty exampleExp)
          print r1
          let r2a = runEval2 (eval2a Map.empty exampleExp)
          print r2a
          let r2b = runEval2 (eval2b Map.empty exampleExp)
          print r2b
          let r2c = runEval2 (eval2c Map.empty exampleExp)
          print r2c
          let r2 = runEval2 (eval2 Map.empty exampleExp)
          print r2
          let r3 = runEval3 Map.empty (eval3 exampleExp)
          print r3
          let r4 = runEval4 Map.empty 0 (eval4 exampleExp)
          print r4
          let r4' = runEval4' Map.empty 0 (eval4' exampleExp)
```

```
    print r4'
    let r5 = runEval5 Map.empty 0 (eval5 exampleExp)
    print r5
    let r5' = runEval5 Map.empty 0 (eval5 (Var "x"))
    print r5'
    r6 <- runEval6 Map.empty 0 (eval6 exampleExp)
    print r6
    r6' <- runEval6 Map.empty 0 (eval6 (Var "x"))
    print r6'
```

zad.hs

```
import Control.Monad.Identity
import Control.Monad.Except
import Control.Monad.State

--dodajemy + - * /
--funkcjonalosc
--1 obsluga bledow(dzielenie przez 0)
--2 tick( state)- ile wykonano operacji
--generowanie logu z obliczen
--monada

--a) typ dla wyrazen
--b) typ dla wynikow
--c) zwykly ewaulatro funkcyjny
```

--d) konwersja ewulatora na styl monadowy(w monadzie Identity)

--e)Dodanie obslugi bledow

--f) dodanie licznika

--g) daodanie logu

```
--A)
data Expr = Lit Int
          | Add Expr Expr
          | Minus Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
          deriving (Show)
--B)
type Value = Int

--c)

eval :: Expr -> Value
eval (Lit n) = n
eval (Add a b) = (eval a) + (eval b)
eval (Minus a b) = (eval a) - (eval b)
eval (Mul a b) = (eval a) * (eval b)
eval(Div a b) = (eval a) `div`(eval b)
```

```
type Eval1 a = Identity a

runEval1 :: Eval1 a -> a
runEval1 = runIdentity

eval1 :: Expr -> Eval1 Value

eval1 (Lit n) = return n
eval1 (Add a b) = do v1 <- eval1 a
                     v2 <- eval1 b
                     return (v1 + v2)
eval1 (Minus a b) = do v1 <- eval1 a
                                           v2 <- eval1 b
                                           return $ v1 - v2
eval1 (Mul a b) = do v1 <- eval1 a
                                          v2 <- eval1 b
                                          return $ v1 * v2
eval1 (Div a b) = do v1 <- eval1 a
                                          v2 <- eval1 b
                                          return $ v1 `div` v2


type Eval2 a = ExceptT String Identity a

runEval2 :: Eval2 a -> Either String a
```

```
runEval2 = runIdentity . runExceptT


eval2 :: Expr -> Eval2 Value


eval2 (Lit n) = return n
eval2 (Add a b) = do v1 <- eval2 a
                     v2 <- eval2 b
                     return $ v1 + v2
eval2 (Minus a b) = do v1 <- eval2 a
                       v2 <- eval2 b
                       return $ v1 - v2
eval2 (Mul a b) = do v1 <- eval2 a
                     v2 <- eval2 b
                     return $ v1 * v2
eval2 (Div a b) = do v1 <- eval2 a
                     v2 <- eval2 b
                     if v2 == 0
                         then throwError "Dzielenie przez 0"
                         else return $ v1 `div` v2


type Eval3 a =StateT Int ( ExceptT String Identity) a
runEval3 :: Integer -> Eval3 a -> (Either String a, Integer)
runEval3 = runIdentity.runExceptT.runStateT st


eval3 :: Expr -> Eval3 Value
```

```
tick :: (Num s, MonadState s m) => m ()
tick = do st <-get
          put (st + 1)

eval3 (Lit n) = do tick
                   return n
eval3 (Add a b) = do tick
                     v1 <- eval3 a
                     v2 <- eval3 b
                     return $ v1 + v2
eval3 (Minus a b) = do tick
                       v1 <- eval3 a
                       v2 <- eval3 b
                       return $ v1 - v2
eval3 (Mul a b) = do tick
                     v1 <- eval3 a
                     v2 <- eval3 b
                     return $ v1 * v2
eval3 (Div a b) = do tick
                     v1 <- eval3 a
                     v2 <- eval3 b
                     if v2 == 0
                        then throwError "Dzielenie przez 0"
                        else return $ v1 `div` v2
```

ewaulator.hs

```
import Control.Monad.Identity

import Control.Monad.Except

import Control.Monad.State


-- Ewaluator wyrażeń arytmetycznych (+, -, *, /)

-- Funkcjonalność:

-- 1.) Obsługa błędów (dzielenie przez zero)

-- 2.) Licznik operacji arytmetucznych wykonanych w trakcie obliczeń

-- 3.) Generowanie logu z obliczeń


-- A. Typ dla wyrażeń


data Expr = Lit Int | Add Expr Expr | Sub Expr Expr | Mul Expr Expr | Div Expr Expr deriving
(Show,Read)


-- B. Typ dla wyniku


type Value = Int
```

```
-- C. Zwykły ewaluator funkcyjny

eval :: Expr -> Value
eval (Lit a)   = a
eval (Add a b) = eval a + eval b
eval (Sub a b) = eval a - eval b
eval (Mul a b) = eval a * eval b
eval (Div a b) = eval a `div` eval b

-- D. Konwersja ewaluatora na styl monadowy (w monadzie Identity)

type Eval1 a = Identity a

runEval1 = runIdentity

eval1 :: Expr -> Eval1 Value
eval1 (Lit a)   = return a
eval1 (Add a b) = do v1 <- eval1 a
                     v2 <- eval1 b
                     return (v1 + v2)
eval1 (Sub a b) = do { v1 <- eval1 a; v2 <- eval1 b; return (v1 - v2) }
eval1 (Mul a b) = do { v1 <- eval1 a; v2 <- eval1 b; return (v1 * v2) }
eval1 (Div a b) = do { v1 <- eval1 a; v2 <- eval1 b; return (v1 `div` v2) }

-- E. Dodanie obsługi błędów
```

```haskell
type Eval2 a = ExceptT String Identity a

runEval2 = runIdentity . runExceptT

eval2 :: Expr -> Eval2 Value
eval2 (Lit a)   = return a
eval2 (Add a b) = do v1 <- eval2 a
                     v2 <- eval2 b
                     return (v1 + v2)
eval2 (Sub a b) = do { v1 <- eval2 a; v2 <- eval2 b; return (v1 - v2) }
eval2 (Mul a b) = (*) <$> eval2 a <*> eval2 b
eval2 (Div a b) = do v1 <- eval2 a
                     v2 <- eval2 b
                     if v2 == 0
                       then throwError "Dzielenie przez zero."
                       else return (v1 `div` v2)

-- F. Dodanie licznika

type Eval3 a = StateT Int (ExceptT String Identity) a

runEval3 e = runIdentity $ runExceptT $ runStateT e 0

eval3 :: Expr -> Eval3 Value
eval3 (Lit a)   = return a
```

```haskell
eval3 (Add a b) = modify (+1) >> (+) <$> eval3 a <*> eval3 b

eval3 (Sub a b) = modify (+1) >> (-) <$> eval3 a <*> eval3 b

eval3 (Mul a b) = modify (+1) >> (*) <$> eval3 a <*> eval3 b

eval3 (Div a b) = do modify (+1)

                     v1 <- eval3 a

                     v2 <- eval3 b

                     if v2 == 0

                       then throwError "Dzielenie przez zero."

                       else return (v1 `div` v2)


-- G. Dodanie logu
```