```
--Thomas Alessandro Buse; 192959 Übung 19
--Paul Rüssmann 196683, Gruppe
{-# LANGUAGE LambdaCase#-}
import Expr
import Control.Monad

--Blatt 11
--Aufgabe 1

type BStore x = x -> Bool
bexp2store :: BExp x -> Store x -> BStore x -> Bool
bexp2store (True_) _ = return True
bexp2store (False_) _ = return False
bexp2store (BVar x) _ = ($x)
bexp2store (Or bs) st = do a <- mapM (\x -> Main.bexp2store x st) bs; return $ (or) a
bexp2store (And bs) st = do b <- mapM (\x -> Main.bexp2store x st) bs ; return $ (and) b
bexp2store (Not bs) st = do c <- Main.bexp2store bs st; return $ (not) c
bexp2store (e1 := e2) st  = return $ Main.exp2store e1 == Main.exp2store e2
bexp2store (e1 :<= e2) st  = return $ Main.exp2store e1 <= Main.exp2store e2


exp2store :: Exp x -> Store x -> Int
exp2store (Con i) = return i
exp2store (Var x) = ($x)
exp2store (Sum es) = do is <- mapM Main.exp2store es; return $ sum is
exp2store (Prod es) = do is <- mapM Main.exp2store es; return $ product is
exp2store (e :- e') = do i <- Main.exp2store e; k <- Main.exp2store e';return $ i-k
exp2store (i :* e)  = do k <- Main.exp2store e; return $ i*k
exp2store (e :^ i)  = do k <- Main.exp2store e; return $ k^i
```

```
--Aufgabe 2
type ID = Int
type Bank = [(ID,Account)]
data Account = Account { balance :: Int, owner :: Client } deriving Show
data Client = Client{ name :: String, surname :: String, address :: String} deriving Show

own1, own2, own3 :: Client
own1 = Client "Max" "Mustermann" "Musterhausen"
own2 = Client "John" "Doe" "Somewhere"
own3 = Client "Erika" "Mustermann" "Musterhausen"

acc1, acc2, acc3 :: Account
acc1 = Account 100 own1
acc2 = Account 0 own2
acc3 = Account 50 own3

bank :: Bank
bank = [(1,acc1), (2,acc2), (3,acc3)]

updRel :: Eq a => [(a,b)] -> a -> b -> [(a,b)]
updRel ((a,b):r) c d = if a == c then (a,d):r else (a,b):updRel r c d
updRel _ a b        = [(a,b)]

credit :: Int -> ID -> Bank -> Bank
credit amount id ls = updRel ls id entry{ balance = oldBalance + amount} where
        Just entry = lookup id ls
        oldBalance = balance entry
debit :: Int -> ID -> Bank -> Bank
debit amount = credit (-amount)
--a)
logMsg :: String -> a -> (String,a)
logMsg message value = (message,value)
```

```
--b)
transferLog :: Int -> ID -> ID -> Bank -> (String, Bank)
transferLog sum idFirst idSecond bank = do
                                                    debit <- logMsg ("Der
Beitrag " ++ (show sum) ++ " wurde von Konto " ++ (show idFirst)) (debit sum idFirst bank)
                                                    credit <- logMsg (" auf
Konto " ++ (show idSecond) ++ " uebertragen.\n") (credit sum idSecond debit)
                                                    return credit
--c)
transactions :: Bank -> (String,Bank)
transactions bank = do
                                        first <- transferLog 50 1 2 bank
                                        second <- transferLog 25 1 3 first
                                        third <- transferLog 25 2 3 second
                                        return third
--Aufgabe 3

newtype State state a = State {runS :: state -> (a,state)}

instance Functor (State state) where
  fmap f (State h) = State $ (\(a,st) -> (f a,st)) . h

instance Monad (State state) where
  return a = State $ \st -> (a,st)
  State h >>= f = State $ (\(a,st) -> runS (f a) st) . h

instance Applicative (State state) where
  pure a = State $ \st -> (a, st)
  stateA2B <*> stateA =  stateA2B >>= flip fmap stateA
--3a)
putAccount :: ID -> Account -> State Bank ()
putAccount id acc = State (\bank -> if (isJust (lookup id bank)) then ((), updRel bank id entry)
```

```
                                                    else
                                                        ((),[(id, acc)]))
                                                        where Just entry =
lookup id bank

--3b)
getAccount :: ID -> State Bank (Maybe Account)
getAccount id = State (\bank -> if (isJust (lookup id bank)) then (Just entry, bank)
                                                    else
                                                        (Nothing,bank))
                                                        where Just entry =
lookup id bank

--3c)
creditS :: Int -> ID -> State Bank ()
creditS amount id = do
                                        account <- getAccount id
                                        if isJust account
                                            then putAccount id (Account (balance (fromJust
account) + amount) (owner (fromJust account)))
                                            else return ()

--3d
debitS :: Int -> ID -> State Bank ()
debitS amount id = do creditS (-amount) id

--3e
transferS :: Int -> ID -> ID -> State Bank ()
transferS sum idFirst idSecond = do
                                                    debitS sum idFirst
                                                    creditS sum idSecond
```