Written Assignment 3

CS 538, Spring 2020

Vinay Patil

1 Practicing do-notation (15)

1.1 Using >>=

(a)
```
func = (Right 17) >>= (\first ->
            return [first, 2] >>= (\second ->
                return (7:second)))
```

```
=> Right [7,17,2]
```

(b)
```
func = MkWithLog (10 - 1, "Subtract 1") >>= (\first ->
                MkWithLog (first*10, "Times n"))
```

```
=> MkWithLog (90,"Subtract 1Times n")
```

(c)
```
func = (if 3 /= 0 then Just (1 / 3) else Nothing) >>= (\first ->
                return (4 * first))
```

```
=> Just 1.3333333333333333
```

1.2 Using >>

(a)
```
func = return "hello" >> Nothing >> Just "world"
```

```
=> Nothing
```

(b)
```
func n = MkWithLog (n, "First") >> MkWithLog (37, "Second") >> return 12
```

```
=> MkWithLog (12,"FirstSecond")
```

1.3 Nested do-blocks

Translate the following examples using >>=, eliminating the do-notation. Simplify as far as you can.

(a)
```
func = Just 3 >>= (\first ->
                Just (first - 7) >>= (\second ->
                    Just (second * 3)))
```

```
=> Just (-12)
```

(b)
```
func = Right [0,3,2,1] >>= (\first ->
                  case first of
                      [] -> Left "empty list"
                      (n:_) -> if n/=0
                               then Right (1/n)
                               else Left "division by zero")

=> Left "division by zero"
```

1.4 More simplifying

(a)
```
func = (if True then MkWithLog (5, "First") else MkWithLog (37, "Second")) >> (return 12)

      -- Since boolean condition is always True, the express in then is selected for evaluation

      = (MkWithLog (5, "First")) >> (return 12)

      -- This will be basically translated to (MkWithLog (5, "First")) >>= (\_ -> return 12)

      -- Also (return 12) lazily evaluates to MkWithLog (12,"")

      = MkWithLog (12, "First" ++ "") -- 5 is ignored by the definition of >>

      -- Final Result will be

      = MkWithLog (12, "First")
```

(b)
```
func = MkWithLog (5, "badger") >> return 4 >> return 3

      -- Evaluation is from left to right

      -- After lazily evaluating (return 4) we get MkWithLog (4, "")

      = MkWithLog (4, "badger" + "") >> return 3

      = MkWithLog (4, "badger") >> return 3

      -- After lazily evaluating (return 3) we get MkWithLog (3, "")

     = MkWithLog (3, "badger" + "")

     -- Final Result will be

     = MkWithLog (3, "badger")
```

(c)
```
func = (Right [4, 7, 6]) >>= (\first ->
                  Left "fail 1" >> Left "fail 2" >> (Right $ (\x -> 9:first)))

      -- Evaluation is from left to right

      -- first takes the value of [4, 7, 6]
```

```
-- But when Haskell see an error case which is Left "Fail 1", it stops executing further

-- Will halt execution and the result value will be

= Left "fail1"
```

2 Lazy datatypes (15)

2.1 Lazy products

(a)

$$\frac{e \to e'}{fst(e) \to fst(e')}$$

$$\frac{}{fst(e1, e2) \to e1}$$

$$\frac{e \to e'}{snd(e) \to snd(e')}$$

$$\frac{}{snd(e1, e2) \to e2}$$

(b) **Haskell Code** :

```
loopForever :: Bool
loopForever = loopForever
```

Lamdba Translation :

$$fix\ x.\ x$$

Step Rule:

$$\frac{}{fix\ x.\ x \to x[x \mapsto (fix\ x.\ x)]}$$

Step:

$$fix\ x.\ x \to fix\ x.\ x$$

Haskell Code :

```
getFst :: Bool
getFst = fst (True, loopForever)
```

Lamdba Translation :

$$fst(true,\ fix\ x.\ x)$$

Step Rule:

$$\frac{}{fst(true,\ fix\ x.\ x) \to true}$$

Step:

$$fst(true,\ fix\ x.\ x) \to true$$

Haskell Code :

```
getSnd :: Bool
getSnd = snd (True, loopForever)
```

Lamdba Translation :

$$snd(true,\ fix\ x.\ x)$$

Step Rule:

$$\frac{}{snd(true,\ fix\ x.\ x) \to fix\ x.\ x}$$

Step:

$$snd(true,\ fix\ x.\ x) \to fix\ x.\ x$$

Then step of loopForever ensures that getSnd never terminates.

2.2 Lazy lists

(a)

$$\frac{e \to e'}{case\ e\ of\ \{nil \to e_1; cons(x_1, x_2) \to e_2\} \to case\ e'\ of\ \{nil \to e_1; cons(x_1, x_2) \to e_2\}}$$

$$\frac{}{case\ nil\ of\ \{nil \to e_1; cons(x_1, x_2) \to e_2\} \to e[x \mapsto nil]}$$

$$\frac{}{case\ cons(e_1, e_2)\ of\ \{nil \to e_1; cons(x_1, x_2) \to e_2\} \to e[x \mapsto cons(e_1, e_2)]}$$

(b) Consider the following bit of Haskell code using lazy lists:

Haskell Code:

```
allTrue :: [Bool]
allTrue = True : allTrue
```

Lamdba Translation:

$$fix\ f.\ \lambda\ x.cons(true, f)$$

Step:

$$fix\ f.\ \lambda\ x.cons(true, f) \rightarrow \text{ cons(true, fix f. } \lambda\ x.cons(true, f)$$

(This is of the form cons(e1,e2) which is a value and hence terminates.)

Haskell Code:

```
negate :: Bool -> Bool
negate b = if b then False else True
```

Lamdba Translation:

$$\lambda\ x.\ (if\ x\ then\ false\ else\ true)$$

Haskell Code:

```
myMap :: (Bool -> Bool) -> [Bool] -> [Bool]
myMap f ls = case ls of
                [] -> []
                (b:bs) -> (f b) : (myMap f bs)
```

Lamdba Translation:

$$fix\ f.\ \lambda\ F.\ \lambda\ L.\ case\ L\ of\ \{nil \rightarrow nil; cons(x_1, x_2) \rightarrow cons((F\ x_1), (f\ F\ x_2))\}$$

Haskell Code:

```
isEmpty :: Bool
isEmpty = case (myMap negate allTrue) of
             [] -> True
             (b:bs) -> False
```

Lamdba Translation:

$$case\ e\ of\ \{nil \rightarrow true; cons(x_1, x_2) \rightarrow false\}$$

Step:

$$case\ e\ of\ \{nil \rightarrow true; cons(x_1, x_2) \rightarrow false\} \rightarrow false$$

(Since it steps to false which is a value, it terminates.)

(More explanation below along with last part.)

Haskell Code:

```haskell
myAccMap :: (Bool -> Bool) -> [Bool] -> [Bool]
myAccMap f ls = go f ls []
    where go f' ls' acc = case ls' of
                              [] -> acc
                              (b:bs) -> go f' bs (acc ++ [f' b])

isAccEmpty :: Bool
isAccEmpty = case (myAccMap negate allTrue) of
                 [] -> True
                 (b:bs) -> False
```

No, the isAccEmpty function would **not terminate**. Unlike the myMap where the bool-function gets applied to the current head in every recursion call, the myAccMap has to wait for the accumulation to finish before the bool-function can be applied, which is forever. This reasoning is similar to why right-fold works on infinite lists and why left-fold doesn't.