```haskell
module Homework2 where

{---------------------- Exercise 1 -------------------------}

type Prog = [Cmd]

data Cmd = LD Int
         | ADD
         | MULT
         | DUP
         deriving Show

type Stack = [Int]

type D = Maybe Stack -> Maybe Stack

--semCmd :: Cmd -> Stack -> Stack
semCmd :: Cmd -> D
semCmd (LD a) xs = case xs of Just xs         -> Just ([a] ++ xs)
                              _               -> Nothing
semCmd (ADD)  xs = case xs of Just (x1:x2:xs) -> Just ([x1+x2] ++ xs)
                              _               -> Nothing
semCmd (MULT) xs = case xs of Just (x1:x2:xs) -> Just ([x1*x2] ++ xs)
                              _               -> Nothing
semCmd (DUP) xs  = case xs of Just (x1:xs)    -> Just ([x1,x1] ++ xs)
                              _               -> Nothing

--sem :: Prog -> Stack -> Stack
sem :: Prog -> D
sem [] a = a
sem (x:xs) a = sem xs (semCmd x a)

eval :: Prog -> Maybe Stack
eval p = sem p (Just [])

--Test data
test1 = [LD 3, DUP, ADD, DUP, MULT] -- [3] -> [3,3] -> [6] -> [6,6] -> Just [36]
test2 = [LD 3, ADD] -- Nothing
test3 = [] -- Just []
test4 = [LD 2, DUP, MULT] -- [2] -> [2, 2] -> Just [4]
-- test5 = [DUP] -- Nothing
-- test6 = [LD 3, LD 8, ADD, LD 5, MULT] -- Just [55]
-- test7 = [LD 3, MULT] -- Nothing

{---------------------- Exercise 2 -------------------------}
```

```
{--
data Cmd2 = C Cmd
          | DEF String Prog
          | CALL String
            deriving Show

type Prog2  = [Cmd2]
type Macros = [(String, Prog)]
type State = (Macros, Maybe Stack)

-- PorM types are a union of Prog2 and Macros
data PorM = P Prog2
          | M Macros
          deriving Show

type E = Maybe State -> Maybe State

-- Define the sematics of a program
sem2 :: Prog2 -> E
sem2 [] a = a
sem2 (x:xs) a = sem2 xs (semCmd2 x a)


-- Define the semantics of Cmd2s
semCmd2 :: Cmd2 -> E
-- Shellout to previous sem function
semCmd2 (C c)       (m, st) = m, semCmd (C c) st
semCmd2 (Def cmd p) (m, st) = [(cmd, p)] ++ m, st
semCmd2 (Call cmd)  (m, st) = semCmd2 cmd (m, st)
-- New commands
semCmd2 c           (m, st) = (m, map semCmd2 nCmd)
                              where nCmd = snd (c, prog):m

-- Evaluate a Prog2 type
--eval2 :: PorM -> Macros
--eval2 p = sem2 p (Just [])

mtest1 = [DEF "foo" [DUP, ADD, MULT]] -- [("foo", [DUP, ADD, MULT])]
--}
{---------------------- Exercise 3 -------------------------}

data Cmd3 = Pen Mode
          | MoveTo Int Int
          | Seq Cmd3 Cmd3
          deriving Show
```

```haskell
          deriving Show

data Mode = Up | Down
          deriving (Show, Eq)

type State = (Mode,Int,Int)
type Line = (Int,Int,Int,Int)
type Lines = [Line]

semS :: Cmd3 -> State -> (State,Lines)
-- Change state if pen was up and now down, or vise versa.
--  Otherwise keep state, produce no lines.
semS (Pen m1)       s@(m2, x, y) | m1 /= m2             = ((m1, x, y), [])
                                 | otherwise            = (s, [])
-- If moving to new position and pen is Down, a line is created,
--  Otherwise keep state, and produce no lines.
semS (MoveTo x1 y1) (m, x2, y2)  | m == Up              = (ns, [])
                                 | x1 /= x2 && y1 /= y2 = (ns, [(x2, y2, x1, y1)])
                                 | otherwise            = (ns, [])
                                 where ns = (m, x1, y1)
semS (Seq a b) s = (fst s2, snd s1 ++ snd s2)
                 where
                    s1 = semS a s
                    s2 = semS b (fst s1)

-- lsem1 = semS (Pen Down) (Up, 0, 0) -- ((Down, 0, 0) [])
-- lsem2 = semS (Pen Up) (Down, 0, 0) -- ((Down, 0, 0) [])
-- lsem3 = semS (MoveTo 2 3) (Up, 0, 0) -- ((Up, 2, 3), [])
-- lsem4 = semS (MoveTo 2 3) (Down, 1, 1) -- ((Down, 2, 3), [(1, 1, 2, 3)])
-- lsem5 = semS ((MoveTo 1 1) `Seq` (MoveTo 2 2)) (Up, 0, 0) -- ((Up 2, 2), [])

-- Initial State
sinit = (Up, 0, 0)

sem' :: Cmd3 -> Lines
-- Keeps track of lines created
sem' a = snd (semS a sinit)

ltest1 = Pen Down `Seq` MoveTo 1 1
ftest1 = sem' ltest1 -- [(0, 0, 1, 1)]
ltest2 = Pen Down `Seq` MoveTo 1 1 `Seq` MoveTo 3 5
ftest2 = sem' ltest2 -- [(0, 0, 1, 1), (1, 1, 3, 5)]
```