```
---------------------------------
--CS381 HW2
---------------------------------
--Yaonan Zhong
--Xiangyu Wang
--Xiaowei Zhang
---------------------------------
--April 29, 2014
---------------------------------

-------------------------------------------------
--Exercise 1. A Stack Language
-------------------------------------------------
type Prog = [Cmd]

data Cmd = LD Int
         | ADD
         | MULT
         | DUP
         deriving (Show, Eq)

type Stack = [Int]
type D = Stack -> Stack

--Define semCmd :: Cmd -> Stack -> Stack
semCmd :: Cmd -> D
semCmd (LD i) xs = [i]++xs
semCmd ADD xs = errTest ADD xs
semCmd MULT xs = errTest MULT xs
semCmd DUP xs = errTest' DUP xs

--Check the length of stack for ADD and MULT
errTest :: Cmd -> D
errTest f xs
      | length xs>=2 = if f==ADD then [a+b]++c else [a*b]++c
      | otherwise = error "less than two elements"
      where (a:b:c) = xs

--Check the length of stack for DUP
errTest' :: Cmd -> D
errTest' f xs
       | length xs>=1 = [a]++xs
       | otherwise = error "less than one element"
       where a:_ = xs

--Call a list of commands
sem :: Prog -> D
sem [] c = c
sem (o:os) c = sem os (semCmd o c)
```

```haskell
----------------------------
--Test
----------------------------
op = [LD 3,DUP,ADD,DUP,MULT]
opp = [LD 3,ADD]
hi = sem op [1,2,3,4,5]



-------------------------------------------------------------------------
--Exercise 2. Extend the Stack Language by Macros
-------------------------------------------------------------------------

data Cmd = LD Int
         | ADD
         | MULT
         | DUP
         | DEF Name Prog
         | CALL Name
         deriving (Show, Eq)

type Name = String
type Prog = [Cmd]
type Macros = [(Name,Prog)]
type Stack = [Int]
type State = (Macros, Stack)
type D = Stack -> Stack

-------------------------------------------------------------------------
--Define sem1 :: [Cmd] -> Stack -> Stack
-------------------------------------------------------------------------
sem1 :: Prog -> D
sem1 os sk = foldl (\acc o -> semCmd o acc) sk os

semCmd :: Cmd -> D
semCmd (LD i) c = [i]++c
semCmd ADD c = errTest ADD c
semCmd MULT c = errTest MULT c
semCmd DUP c = errTest' DUP c

--Check the length of stack for ADD and MULT
errTest :: Cmd -> D
errTest f xs
     | length xs>=2 = if f==ADD then [a+b]++c else [a*b]++c
     | otherwise = error "less than two elements"
     where (a:b:c) = xs

--Check the length of stack for DUP
errTest' :: Cmd -> D
errTest' f xs
```

```
      | length xs>=1 = [a]++xs
      | otherwise = error "less than one element"
      where a:_ = xs

-----------------------------------------------------------------------
--Define sem2 :: Cmd -> State -> Stack
-----------------------------------------------------------------------
type DD = State -> Stack
sem2 :: Cmd -> DD
sem2 ca (ma,sk) = sem1 (errMac ca ma) sk

--Check whether the function called is in the definition
errMac :: Cmd -> Macros -> Prog
errMac (CALL n) [] = error "no match macros"
errMac (CALL n) ((na,pr):xs)
    | n==na = pr
    | otherwise = errMac (CALL n) xs

-------------------------------------------------------------------
--Define sem3 :: [Cmd] -> State -> Stack
-------------------------------------------------------------------

--Call a list of commands
sem3 :: Prog -> DD
sem3 cas (ma,sk) = foldl (\acc ca -> sem2 ca (ma,acc)) sk cas

---------------------------------
--Test
---------------------------------
sqr = [DUP,MULT]
mac = [("SQR",sqr)]
callsqr = CALL "SQR"
stk = [2,2,3,4]
emp = []

sq = sem2 callsqr (mac,stk)
err1 = sem2 callsqr ([("abc",sqr)],stk)
err2 = sem2 callsqr (mac,emp)

newsqr = [LD 4]++sqr
newmac = mac++[("NEWSQR",newsqr)]
sq4 = sem2 (CALL "NEWSQR") (newmac,stk)

sqsq = sem3 [CALL "NEWSQR",CALL "SQR"] (newmac,stk)
```

```
---------------------------------
--Exercise 3. Mini Logo
---------------------------------
import SVG (ppLines)

data Cmd = Pen Mode
         | MoveTo Int Int
         | Seq Cmd Cmd
         deriving Show

data Mode = Up|Down deriving Show

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

semS :: Cmd -> State -> (State,Lines)
semS (Pen m) (a,b,c) = ((m,b,c),[(b,c,b,c)])
semS (MoveTo i j) (a,b,c) = ((a,i,j),[(b,c,i,j)])
semS (Seq s k) st = (fst kl, snd sl++snd kl)
                  where sl = semS s st
                        kl = semS k (fst sl)


sem' :: Cmd -> Lines
sem' c = snd (semS c (Up,0,0))

-----------------------------------
--Test
-----------------------------------
l = semS (Seq (Seq (Pen Down) (MoveTo 1 1)) (MoveTo 2 2)) (Up,0,0)
prog = Seq(Seq (Seq (Pen Down) (MoveTo 1 1)) (MoveTo 2 2))(MoveTo 2 5)
pp = ppLines (sem' prog)
```