```haskell
module Homework3 where

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

type Prog = [Cmd]

data Cmd = LD Int
         | ADD
         | MULT
         | DUP
         | INC
         | SWAP
         | POP Int
         deriving Show

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

type Rank = Int
type CmdRank = (Int, Int)


-- Semantics of individual Commands
semCmd :: Cmd -> D
semCmd (LD a)  xs         = [a] ++ xs
semCmd (ADD)   (x1:x2:xs) = [x1+x2] ++ xs
semCmd (MULT)  (x1:x2:xs) = [x1*x2] ++ xs
semCmd (DUP)   (x1:xs)    = [x1,x1] ++ xs
semCmd (INC)   (x1:xs)    = [succ x1] ++ xs
semCmd (SWAP)  (x1:x2:xs) = (x2:x1:xs)
semCmd (POP n) xs         = drop n xs
semCmd _       _          = []

-- Semantics of a Program
sem :: Prog -> D
sem [] a = a
sem (x:xs) a = sem xs (semCmd x a)

-- Assign ranks to commands
```

```
--
rankC :: Cmd -> CmdRank
rankC (LD _)  = (0, 1)
rankC ADD     = (2, 1)
rankC MULT    = (2, 1)
rankC DUP     = (1, 2)
rankC INC     = (1, 1)
rankC SWAP    = (2, 2)
rankC (POP a) = (a, 0)

rankP :: Prog -> Maybe Rank
rankP xs = rank xs 0

-- Rank a program
--
rank :: Prog -> Rank -> Maybe Rank
rank []     r | r >= 0     = Just r
rank (x:xs) r | under >= 0 = rank xs (under+adds)
              where (subs, adds) = rankC x
                    under        = r - subs
rank _      _ = Nothing

{--- Part (b) ---}

data Type = A Stack | TypeError deriving Show

typeSafe :: Prog -> Bool
typeSafe p = (rankP p) /= Nothing

semStatTC :: Prog -> Type
semStatTC p | typeSafe p = A (sem p [])
            | otherwise  = TypeError
{-
  Question:
      What is the new type of the function sem and why can the
      function definition be simplified to have this type?

  Answer:
       The new type of sem is 'Prog -> D' where type D = Stack -> Stack.
       type D can be simplified to no longer contain Maybe Stacks
```

```haskell
        type D can be simplified to no longer contain Maybe Stacks,
        because the type checker handles all TypeErrors.
-}

p1 = [LD 3, DUP, ADD, LD 5, SWAP] -- A [6, 5]
p2 = [LD 8, POP 1, LD 3, DUP, POP 2, LD 4] -- A [4]
p3 = [LD 3, LD 4, LD 5, MULT, ADD] -- A [23]
p4 = [LD 2, ADD] -- TypeError
p5 = [DUP] -- TypeError
p6 = [POP 1] -- TypeError


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

data Shape = X
           | TD Shape Shape
           | LR Shape Shape
           deriving Show

type BBox = (Int, Int) -- (width, height) of bounding box

{- (a) Define a type checker for the shape language -}
--
bbox :: Shape -> BBox
bbox (TD i j) -- width is that of the wider one; height is sum of heights
    | ix >= jx = (ix, iy + jy)
    | ix < jx = (jx, iy + jy)
    where (ix, iy) = bbox i
          (jx, jy) = bbox j
bbox (LR i j) -- width is sum of widths; height is that of the taller one
    | iy >= jy = (ix + jx, iy)
    | iy < jy = (ix + jx, jy)
    where (ix, iy) = bbox i
          (jx, jy) = bbox j
bbox X = (1, 1)

{- (b) Define a type checker for the shape language that assigns
        types only to rectangular shapes -}

rect :: Shape -> Maybe BBox
```

```haskell
rect X = Just (1, 1)
rect (TD i j) =             -- widths must match, and bbox has that width and
    case rect i of          -- its height is sum of heights. Else Nothing.
        Nothing -> Nothing
        Just (ix, iy) -> case rect j of
                             Nothing -> Nothing
                             Just (jx, jy) -> case (ix == jx) of
                                                  True -> Just (ix, iy + jy)
                                                  False -> Nothing
rect (LR i j) =             -- heights must match, and bbox is that height
    case rect i of          -- with width the sum of widths. Else Nothing.
        Nothing -> Nothing
        Just (ix, iy) -> case rect j of
                             Nothing -> Nothing
                             Just (jx, jy) -> case (iy == jy) of
                                                  True -> Just (ix + jx, iy)
                                                  False -> Nothing

-- Test Shapes
r1 = TD (LR X X) (LR X X) -- bbox (2,2), rect Just (2,2)
r2 = TD (LR X X) X -- bbox (2,2), rect Nothing
r3 = LR (TD r1 X) (LR r2 r2) -- bbox (6, 3), rect Nothing
r4 = LR (TD r1 r1) (TD r1 r1) -- bbox (4, 4), rect Nothing
r5 = LR r4 r4 -- bbox (8, 4), rect Just (8, 4)

{---------------------- Exercise 3 ------------------------}

{- (a) Consider the functions f and g, which are given by the
    following two function definitions.  -}

f x y = if null x then [y] else x
g x y = if not (null x) then [] else [y]

{- (1) What are the types of f and g?
        f :: [a] -> a -> [a]
        g :: [a] -> b -> [b]

   (2) Explain why the functions have these types.
        Since f will return either [y] or x, and x is a list, the elements
        of x have to be of the same type as y. This is because, to the
```

```haskell
        of x have to be of the same type as y. This is because, to the
        best of our knowledge) Haskell can't return two different types
        from a function.

        While similar to f, g will return either [] or [y]. The subtle
        difference here is that y now has no relation to x, since a list
        is a phantom type. This make Haskell assume the second argument
        to g is not the same type as the first.

    (3) Which type is more general?
        Because both f and g will work with any types they are both
        general, but one could make the argument that because g works
        with more than one type, it is more general.

    (4) Why do f and g have different types?
        f and g have different types because of the magic of Haskell type
        inference.
-}

{- (b) Find a (simple) definition for a function h that has the
       following type. -}

h :: [b] -> [(a, b)] -> [b]
h b _ = b

{- (c) Find a (simple) definition for a function k that has the
        following type.

    k :: (a -> b) -> ((a -> b) -> a) -> b

    We can not find a (simple) definition for function k, as there is no
    way in Haskell to pattern match a function and its parameters at the
    same time. Also since the function signature only defines b in the
    terms of being the return type of another function, we can not deduce
    anything about how b should be represented.

   (d) Can you define a function of type a -> b?
       If yes, explain your definition. If not, explain why it is
       so difficult.

```

```
        No. Defining a function of type a -> b requires knowing something
        about type b. Since we don't have that knowledge, we can not
        define how something of type b should be represented. Anything we
        might use would end up restricting what b might be, thus it would
        not be of any type.

        We could write:
            j :: a -> b
            j = j

        But this is a circular definition and will never terminate, thus
        we have not truly defined anything at all.

-}
```