```haskell
#!/usr/bin/env/stack
-- stack --install-ghc runghc

-- CS 381 HW 3
-- Prof Erwig

-- Authors:
-- Jackson Neff
-- Rhea Mae Edwards

main :: IO()
main = putStrLn("Program Executed.")

--exercise 1

--- part a ---

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

type Stack = [Int]

type Prog = [Cmd]

type D = Stack -> Stack

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

semCmd :: Cmd -> D
semCmd (LD a) xs = ([a] ++ xs)

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

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



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

rank :: Prog -> Rank -> Maybe Rank
rank []      s | s >= 0     = Just s
```

```
rank (x:xs)  s | under >= 0 = rank xs (under+push)
               where (pop, push) = rankC x
                     under       =  s - pop
rank _       _ = Nothing


---- Part b ----

data Type = S Stack | TypeError deriving Show

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

semStatTC :: Prog -> Type
semStatTC p | typeSafe p  = S(sem p [])
            | otherwise   = TypeError



-- Part 2 --

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

type BBox = (Int, Int)

--- 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)

rect :: Shape -> Maybe BBox
rect X = Just(1, 1)
rect (TD i j) =
    case rect i of
        Nothing -> Nothing                --- If i is not a rectangle then return nothing ---
        Just (ix, iy) -> case rect j of   --- If i is a rectangle then check j ---
                         Nothing -> Nothing
                         Just (jx, jy) -> case (ix == jx) of  --- If j is a rectangle check if
                         i is the same width ---
                                          True -> Just (ix, iy + jy)  --- If it is same width,
                                          place j on top of i. ---
```

```
                                           False -> Nothing
rect (LR i j) =
    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

-- Part 3 --

-- 1 --
-- a. What types are f and g?
-- Function f is of type [a] -> [a] -> [a]
-- Function g is of type [a] -> [b] -> [b]

-- b. Why do the functions have these types?
--f has this type because it takes two lists and outputs one, and both lists must have the same
-- type because either x or y can be returned and a Haskell function can have at most 1 return
type.
-- g has this type because it can return either an untyped list (empty list) or a list of type y.
-- the type of x does not matter because it will not be returned no matter what.

-- c. Which type is more general?
-- g has a more general type, because there are less restrictions. in f, all three lists must
be of the same type,
-- but in g, the type of x doesnt matter, allowing for more flexibility

-- d. Why do f and g have different types?
-- They need to. Although any valid input to f can also be a valid input to g, we still must
specify the
-- restrictions of f. Mostly this means denoting that all list types in f must be the same, but
the only
-- two that need to be the same in g are parameters 2 and 3.

-- 2 --

h :: [b] -> [(a,b)] -> [b]
h b (x:xs) = [snd(x)] ++ b;
h b _ = b

-- 3 --
--k :: (a -> b) -> ((a -> b) -> a) -> b   --Order of parentheses should not matter, I think?
--k f x = func(f x)



-- 4 --

-- No, you cannot make this defnition for a function.
-- To convert one type to another, you must know all possible values of the type you are
converting to, in this case b.
-- Unless you know all those values and develop case statements that only allow valid output,
```

```
you will end
-- up restricting type b with some operation related to a. For example, if the function added 5
to everything, then a would
-- have to be an numerical type, but then so would b because by adding to a, you restrict the
output, b, to numerical types.
-- In any event, b would be the same type as a and the definition would fail, as it stipulates
a and b are different types.
```