```haskell
{-# LANGUAGE GADTs #-} -- used in testing infrastructure
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}
module Hw08 where

{-

v1.1

Run this file with `stack runghc <this-file>`
Load this file into an interactive prompt with `stack ghci <this-file>`

Name: <put your name here>

Collaboration Statement:

<put your collaboration statement here>

Course Policy Reminder:

    Collaboration with peers on the high-level ideas and approach on
    assignments is encouraged. Copying someone else's work is not allowed. Any
    collaboration, even at a high level, must be declared when you submit your
    assignment. Every assignment must include a collaboration statement. E.g.,
    “I discussed high-level strategies for solving problem 2 and 5 with Alex.”

    Obtaining high-level information on the internet is allowed and encouraged
    if it helps you learn the material. However, I strongly discourage googling
    for answers to homework problems. Copying code from the internet and
    submitting copied content for assignments is not allowed.

    Students caught copying work from peers or submitting copied code from the
    internet will be eligible for immediate failure of the course and
    disciplinary action by the University. All academic integrity misconduct
    will be treated according to UVM's Code of Academic Integrity.

-}

import Data.Map (Map)
import qualified Data.Map as Map

import Data.List          -- used in testing infrastructure
import Control.Monad      -- used in testing infrastructure
import Control.Exception  -- used in testing infrastructure
import System.IO          -- used in testing infrastructure

-- x ∈ var ≈ symbol
-- e ⩴ i
--   | e + e
--   | b
--   | IF e THEN e ELSE e
--   | x
--   | FUN (x:τ) ⇒ e
--   | e(e)
data Expr = IntE Integer
          | PlusE Expr Expr
          | BoolE Bool
          | IfE Expr Expr Expr
          | VarE String
          | LetE String Expr Expr
          | FunE String Type Expr
          | AppE Expr Expr
  deriving (Eq,Ord,Show)

-- v ∈ value ⩴ i
--           | b
--           | ⟨FUN (x) ⇒ e,γ⟩
data Value = IntV Integer
           | BoolV Bool
           | FunV String Expr VEnv
  deriving (Eq,Ord,Show)

-- γ ∈ venv ≜ var ⇀ value
type VEnv = Map String Value

-- τ ∈ type ⩴ int
--          | bool
--          | τ ⇒ τ
data Type = IntT
          | BoolT
          | ArrowT Type Type
  deriving (Eq,Ord,Show)

-- Γ ∈ tenv ≜ var ⇀ type
type TEnv = Map String Type

-- interp ∈ venv × expr ⇀ value
-- interp(γ,i) ≜ i
-- interp(γ,e₁ + e₂) ≜ i₁ + i₂
--   where i₁ = interp(γ,e₁)
--         i₂ = interp(γ,e₂)
-- interp(γ,b) ≜ b
-- interp(γ,IF e₁ THEN e₂ ELSE e₃) ≜
--   CASES
--     interp(γ,e₂) IF true = interp(γ,e₁)
--     interp(γ,e₃) IF false = interp(γ,e₁)
-- interp(γ,x) ≜ γ(x)
-- interp(γ,FUN(x)⇒e) ≜ ⟨FUN(x)⇒e,γ⟩
-- interp(γ,e₁(e₂)) ≜ interp(γ′[x↦v],e′)
--   where ⟨FUN(x)⇒e′,γ′⟩ = interp(γ,e₁)
--         v = interp(γ,e₂)
interp :: VEnv -> Expr -> Maybe Value
interp env e0 = case e0 of
  IntE i -> Just (IntV i)
  PlusE e1 e2 -> case (interp env e1,interp env e2) of
    (Just (IntV i1),Just (IntV i2)) -> Just (IntV (i1 + i2))
    _ -> Nothing
  BoolE b -> Just (BoolV b)
  IfE e1 e2 e3 -> case interp env e1 of
    Just (BoolV b) ->
      if b
      then interp env e2
      else interp env e2
    _ -> Nothing
  VarE x -> Map.lookup x env
  LetE x e₁ e₂ -> case interp env e₁ of
    Just v₁ -> interp (Map.insert x v₁ env) e₂
    _ -> Nothing
  FunE x _ e -> Just (FunV x e env)
  AppE e1 e2 -> case (interp env e1,interp env e2) of
    (Just (FunV x e' env'),Just v) -> interp (Map.insert x v env') e'
    _ -> Nothing

-- [E1] ★★★
-- Complete the eight missing cases of this type checker for a core language
-- with integers, booleans, let-binding and functions.
--
-- check ∈ venv × expr ⇀ type
-- check(Γ,e) = τ ⇔ Γ ⊢ e : τ
--   where
--
--   -----------
--   Γ ⊢ i : int
--
--   Γ ⊢ e₁ : int
--   Γ ⊢ e₂ : int
--   ------------
--   Γ ⊢ e₁ + e₂ : int
--
--   -----------
--   Γ ⊢ b : bool
--
--   Γ ⊢ e₁ : bool
--   Γ ⊢ e₂ : τ
--   Γ ⊢ e₃ : τ
--   --------------------------------
--   Γ ⊢ IF e₁ THEN e₂ ELSE e₃ : τ
--
--   Γ(x) = τ
--   ----------------
--   Γ ⊢ x : τ
--
--   Γ ⊢ e₁ : τ₁
--   Γ[x↦τ₁] ⊢ e₂ : τ₂
--   -----------------------
--   Γ ⊢ LET x = e₁ IN e₂ : τ₂
--
--   Γ[x↦τ₁] ⊢ e : τ₂
--   --------------------
--   Γ ⊢ FUN (x:τ₁) ⇒ e : τ₁ ⇒ τ₂
--
--   Γ ⊢ e₁ : τ₁ ⇒ τ₂
--   Γ ⊢ e₂ : τ₁
--   ------------------
--   Γ ⊢ e₁(e₂) : τ₂
--
check :: TEnv -> Expr -> Maybe Type
check env e0 = case e0 of
  IntE _ -> error "TODO"
  PlusE e1 e2 -> error "TODO"
  BoolE _ -> error "TODO"
  IfE e1 e2 e3 -> error "TODO"
  VarE x -> error "TODO"
  LetE x e1 e2 -> error "TODO"
  FunE x t e -> error "TODO"
  AppE e1 e2 -> error "TODO"

checkTests :: (Int,String,Expr -> Maybe Type,[(Expr,Maybe Type)])
checkTests =
  (1
  ,"interp"
  ,check Map.empty
  ,[ -- e = 1 + 2
    ( PlusE (IntE 1) (IntE 2)
      -- τ = int
    , Just IntT
    )
   , -- e = 1 + true
    ( PlusE (IntE 1) (BoolE True)
      -- type error
    , Nothing
    )
   , -- e = LET x = 1 IN x + 2
    ( LetE "x" (IntE 1) $
      PlusE (VarE "x") (IntE 2)
      -- τ = int
    , Just IntT
    )
   , -- e = IF true THEN 1 ELSE 2
    ( IfE (BoolE True) (IntE 1) (IntE 2)
      -- τ = int
    , Just IntT
    )
   , -- e = IF true THEN false ELSE true
    ( IfE (BoolE True) (BoolE False) (BoolE True)
      -- τ = bool
    , Just BoolT
    )
   , -- e = IF true THEN 1 ELSE true
    ( IfE (BoolE True) (IntE 1) (BoolE True)
      -- type error
    , Nothing
    )
   ]
  )

-- [X1]
-- EXTRA CREDIT PROBLEM
--
-- add references to the language. use the same language as HW6 with (1) box
-- creation, (2) box access, and (3) box assignment.
--
-- there is one new type
--
--     τ ⩴ … | BOX τ
-- typing rules are:
--
--     Γ ⊢ e : τ
--     -----------------
--     Γ ⊢ BOX e : BOX τ
```

```
--
--     Γ ⊢ e : BOX τ
--     -------------
--     Γ ⊢ !e : τ
--
--     Γ ⊢ e₁ : BOX τ
--     Γ ⊢ e₂ : τ
--     --------------
--     Γ ⊢ e₁ ← e₂ : τ
--
-- Here is an example program that should type check to (BOX int)
--
--     LET b = BOX 1 IN
--     LET _ = b ← (1 + !b) IN
--     b
--
-- Feel free to copy and re-use any code from above, however make sure you do
-- *not* modify any of the code used in E1.

---------------
-- ALL TESTS --
---------------

allTests :: [Test]
allTests =
  [ Test1 checkTests
  ]

----------------------
-- MAIN = RUN TESTS --
----------------------

main :: IO ()
main = runTests allTests

----------------------------
-- TESTING INFRASTRUCTURE --
----------------------------

mapOn :: [a] -> (a -> b) -> [b]
mapOn = flip map

foldMOn :: (Foldable t,Monad m) => b -> t a -> (b -> a -> m b) -> m b
foldMOn i xs f = foldM f i xs

data Test where
  Test1 :: (Show a,Eq b,Show b) => (Int,String,a -> b,[(a,b)]) -> Test
  Test2 :: (Show a,Show b,Eq c,Show c) => (Int,String,a -> b -> c,[((a,b),c)]) -> Test
  Test3 :: (Show a,Show b,Show c,Eq d,Show d) => (Int,String,a -> b -> c -> d,[((a,b,c),d)]) -> Test

runTests :: [Test] -> IO ()
runTests ts = do
  rs <- forM ts $ \ t -> do
    y <- case t of
      Test1 t -> runTests1 t
      Test2 t -> runTests2 t
      Test3 t -> runTests3 t
    putStrLn ""
    return y
  forM_ (zip [1..] rs) $ \ (m,(n,passed,failed)) -> do
    when (m /= 1) $ putStrLn ""
    putStrLn $ "++ E" ++ show n ++ " Tests Passed: " ++ show passed
    putStrLn $ "-- E" ++ show n ++ " Tests Failed: " ++ show failed

showTestResult :: (Eq a,Show a) => String -> a -> Either String a -> (Int,Int) -> IO (Int,Int)
showTestResult fx y y'M (passed,failed) = do
  let eM = case y'M of
        Left e -> Just $ "[ERROR]: " ++ e
        Right y' ->
          if y' == y
          then Nothing
          else Just $ show y'
  case eM of
    Nothing -> do
      putStrLn $ "   [TEST PASSED]: " ++ fx
      hFlush stdout
      return (passed+1,failed)
```

```haskell
    Just s -> do
      putStrLn $ "   [TEST FAILED]:"
      putStrLn $ "     -- the input"
      putStrLn $ "     " ++ fx
      putStrLn $ "   =="
      putStrLn $ "     -- the output"
      putStrLn $ "     " ++ s
      putStrLn $ "   /="
      putStrLn $ "     -- the expected result"
      putStrLn $ "     " ++ show y
      hFlush stdout
      return (passed,failed+1)

runTestsN :: (Eq a,Show a) => Int -> String -> [(String,() -> a,a)] -> IO (Int,Int,Int)
runTestsN n name tests = do
  putStrLn $ ">> E" ++ show n ++ " Tests: " ++ name
  (passed,failed) <- foldMOn (0,0) tests $ \ pf (s,fx,y) -> do
    y'M <- catch (Right <$> evaluate (fx ())) $ \ (SomeException e) -> return $ Left $ chomp $ unwords $ lines $ show e
    showTestResult s y y'M pf
  return (n,passed,failed)
  where
    chomp s0 = concat $ mapOn (group s0) $ \ s ->
      if " " `isPrefixOf` s then " " else s

runTests1 :: (Eq b,Show a,Show b) => (Int,String,a -> b,[(a,b)]) -> IO (Int,Int,Int)
runTests1 (n,name,f,tests) = runTestsN n name $ mapOn tests $ \ (x,y) ->
  (name ++ " " ++ showsPrec 11 x [],\() -> f x,y)

runTests2 :: (Eq c,Show a,Show b,Show c) => (Int,String,a -> b -> c,[((a,b),c)]) -> IO (Int,Int,Int)
runTests2 (n,name,f,tests) = runTestsN n name $ mapOn tests $ \ ((x,y),z) ->
  (name ++ " " ++ showsPrec 11 x [] ++ " " ++ showsPrec 11 y [],\() -> f x y,z)

runTests3 :: (Eq d,Show a,Show b,Show c,Show d) => (Int,String,a -> b -> c -> d,[((a,b,c),d)]) -> IO (Int,Int,Int)
runTests3 (n,name,f,tests) = runTestsN n name $ mapOn tests $ \ ((w,x,y),z) ->
  (name ++ " " ++ showsPrec 11 w [] ++ " " ++ showsPrec 11 x [] ++ " " ++ showsPrec 11 y [],\() -> f w x y,z)
```