```haskell
{-# LANGUAGE GADTs #-}
module Exam.Ex1 where

import Test.QuickCheck
import Control.Applicative
import Data.Char
import Data.Maybe

-- | A simple expression language with integers and booleans.
-- Contains both well- and ill-typed expressions.
data Expr where
  LitI   :: Int  -> Expr
  LitB   :: Bool -> Expr
  (:+:)  :: Expr -> Expr -> Expr
  (:==:) :: Expr -> Expr -> Expr
  If     :: Expr -> Expr -> Expr -> Expr
  -- New ----------------------------------------
  Nil    :: Expr
  Cons   :: Expr -> Expr -> Expr

-- | A value is an integer or a boolean.
data Value = VInt Int
           | VBool Bool
           | VList [Value] -- list of expressions, 5 pts
  deriving (Eq, Show)

showTypeOfVal :: Value -> String
showTypeOfVal (VBool _)  = "Bool"
showTypeOfVal (VInt  _)  = "Int"
-- New ------------------------------------------
showTypeOfVal (VList vs) = "["++ "Value" ++ "]"

-- | Evaluation needs to check the expressions are evaluated to
-- the right type; otherwise, it produces an error.
eval :: Expr -> Value
eval (LitI n)       =  VInt n
eval (LitB b)       =  VBool b

eval (e1 :+: e2) = let v1 = eval e1
                       v2 = eval e2
                   in case (v1,v2) of
                        (VInt n1, VInt n2)   -> VInt  (n1+n2)
                        -- New
                        (VList l1, VList l2) -> sumlist l1 l2
                        _                    -> error "Crash!"
eval (e1 :==: e2) =
   let v1 = eval e1
       v2 = eval e2
   in case (v1, v2) of
     (VList l1, VList l2) -> VBool $ check_eq v1 v2
     (VInt n1 , VInt n2)  -> VBool $ check_eq v1 v2
     (VBool b1, VBool b2) -> VBool $ check_eq v1 v2
     _                    -> error "Crash!"

eval (If e1 e2 e3) =
  case (eval e1) of
       VBool b -> if b then eval e2
                       else eval e3
       _       -> error "Crash!"

-- New ------------------------------------------
eval Nil         = VList []
eval (Cons x xs) = let VList ys = eval xs
                    in VList $ (eval x) : ys

-- New ------------------------------------------
check_eq :: Value -> Value -> Bool
check_eq (VList l1) (VList l2)
```

```haskell
  | length l1 == length l2 = and $ map (uncurry check_eq) (zip l1 l2)
  | otherwise = False
check_eq (VInt n1) (VInt n2)   = n1 == n2
check_eq (VBool b1) (VBool b2) = b1 == b2

-- New -----------------------------------------
sumlist :: [Value] -> [Value] -> Value
sumlist l1 l2
  | length l1 == length l2 = sumVals l1 l2
  | otherwise              = error "I cannot sum lists of different lengths!"
 where sumVals []           []           = VList []
       sumVals (VInt n:ns) (VInt m:ms) = let VList rs = sumVals ns ms
                                          in  VList $ VInt (n+m):rs
       sumVals _            _            = error "Problems adding lists!"

-- | Examples
eOK, eBad, eBad2  :: Expr
eOK   = If (LitB False) (LitI 2) (LitI 2 :+: LitI 1736)
eBad  = If (LitB False) (LitI 1) (LitI 2 :+: LitB True)
eBad2 = If (LitI 20) (LitI 1) (LitI 7)

-- Pretty printing.
instance Show Expr where
  showsPrec p e = case e of
    LitI n       -> shows n

    LitB b       -> shows b

    e1 :+: e2     -> showParen (p > 2) $
      showsPrec 2 e1 . showString " + " . showsPrec 3 e2

    e1 :==: e2    -> showParen (p > 1) $
      showsPrec 2 e1 . showString " == " . showsPrec 2 e2

    If e1 e2 e3  -> showParen (p > 0) $
      showString "if "    . shows e1 .
      showString " then " . shows e2 .
      showString " else " . shows e3
-- New -----------------------------------------
    Nil -> showString "[]"

    Cons e1 e2 -> showString "["  .
                  sLE e1 e2        .
                  showString "]"

-- New -----------------------------------------
sLE e1 Nil             = shows e1
sLE e1 (Cons e1' e2') = shows e1 . showString "," . sLE e1' e2'
sLE _ _ = error "The list got broken"



-- Test cases
test1  = Nil
test2  = Cons (LitI 1) (Cons (LitB True) Nil)
test3  = Cons (LitI 1) (Cons (LitB True) (Cons (LitI 42) Nil))
test4  = test2 :==: test2
test5  = test2 :==: test3
test6  = Cons test2 (Cons test3 Nil)
test7  = test6 :==: test6
test8  = Cons (LitI 1) (Cons (LitI 2) Nil)
test9  = Cons (LitI 100) (Cons (LitI 1) Nil)
test10 = test8 :+: test9
test11 = Cons test8 Nil :+: Cons test9 Nil -- it should fail


---------------------------------
```

```
-- Quick Check Generator
---------------------------------

instance Arbitrary Expr where
  arbitrary = sized arbExpr
-- Exercise1: generate "well-typed" expressions (in ExprB & ExprI)
arbExpr :: Int -> Gen Expr
arbExpr n | n <= 0 = basecases
arbExpr n | otherwise = oneof
    [ basecases
    , (:+:)  <$> arbExpr2 <*> arbExpr2
    , (:==:) <$> arbExpr2 <*> arbExpr2
    , If    <$> arbExpr3 <*> arbExpr3 <*> arbExpr3
    -- New ------------------------------------------
    , Cons  <$> arbExpr2 <*> arbExpr2
    ]
  where arbExpr2 = arbExpr (n `div` 2)
        arbExpr3 = arbExpr (n `div` 3)

basecases :: Gen Expr
basecases = oneof [ LitI <$> arbitrary
                  , LitB <$> arbitrary
                  -- New ------------------------------------------
                  , Cons <$> arbitrary <*> return Nil
                  ]
```

```haskell
{-# LANGUAGE GADTs, ExistentialQuantification, FlexibleInstances #-}
module Exam.Ex2 where

import qualified Exam.Ex1 as Ex1
import Data.Maybe (fromJust, isJust)
import Test.QuickCheck
import Control.Monad

infixl 6 :+:
infix  4 :==:
infix  0 :::

-- | The type of well-typed expressions. There is no way to
-- construct an ill-typed expression in this datatype.
data Expr t where
  LitI   :: Int -> Expr Int
  LitB   :: Bool -> Expr Bool
  -- New ---------------------------------------
  (:+:)  :: Add t => Expr t -> Expr t -> Expr t
  ---------------------------------------------
  (:==:) :: Eq t => Expr t -> Expr t -> Expr Bool
  If    :: Expr Bool -> Expr t -> Expr t -> Expr t
  -- New ----------------------------------------
  -- Task 3.1
  Nil   :: Expr [t]
  Cons  :: Expr t -> Expr [t] -> Expr [t]

class Add t where
  add :: t -> t -> t

instance Add Int where
  add = (+)

instance Add [Int] where
  add e1 e2 | length e1 == length e2 = map (uncurry (+)) (zip e1 e2)
            | otherwise = error "I cannot sum two lists of different lenghts"

-- | A type-safe evaluator. Much nicer now that we now that
-- expressions are well-typed. No Value datatype needed, no
-- extra constructors VInt, VBool.
eval :: Expr t -> t
eval (LitB b)    = b
-- New ----------------------------------------
eval (e1 :+: e2) = add (eval e1) (eval e2)
----------------------------------------------
eval (e1 :==: e2) = eval e1 == eval e2
eval (If b t e)  = if eval b then eval t else eval e
eval (LitI n)    = n
-- New ----------------------------------------
eval Nil = []
eval (Cons t ts) = (eval t):(eval ts)


eOK :: Expr Int
eOK  = If (LitB False) (LitI 1) (LitI 2 :+: LitI 1736)
-- eBad = If (LitB False) (LitI 1) (LitI 2 :+: LitB True)

-- | We can forget that an expression is typed. For instance, to
-- be able to reuse the pretty printer we already have.
forget :: Expr t -> Ex1.Expr
forget e = case e of
  LitI n       -> Ex1.LitI n
  LitB b       -> Ex1.LitB b
  e1 :+: e2    -> forget e1 Ex1.:+: forget e2
  e1 :==: e2   -> forget e1 Ex1.:==: forget e2
  If e1 e2 e3  -> Ex1.If (forget e1) (forget e2) (forget e3)
  -- New ----------------------------------------
  -- Task 3.2
```

```
  Nil          -> Ex1.Nil
  Cons t ts    -> Ex1.Cons (forget t) (forget ts)

instance Show (Expr t) where
  showsPrec p e = showsPrec p (forget e)

-- How to go the other way, turning an untyped expression into a
-- typed expression?


-- | The types that an expression can have. Indexed by the
-- corresponding Haskell type.
data Type t where
  TInt  :: Type Int
  TBool :: Type Bool
  -- New -----------------------------------------
  TList :: Type t -> Type [t]

instance Show (Type t) where
  show TInt      = "Int"
  show TBool     = "Bool"
  -- New -----------------------------------------
  show (TList t) = "[" ++ show t ++ "]"

-- | Well-typed expressions of some type are just pairs of
-- expressions and types which agree on the Haskell type. The
-- /forall/ builds an existential type (exercise: think about
-- whether this makes sense).
data TypedExpr where
  (:::) :: Eq t => Expr t -> Type t -> TypedExpr

instance Show TypedExpr where
  show (e ::: t) = show e ++ " :: " ++ show t

data Equal a b where
  Refl :: Equal a a

-- | The type comparison function returns a proof that the types
-- we compare are equal in the cases that they are.
(=?=) :: Type s -> Type t -> Maybe (Equal s t)
TInt  =?= TInt            = Just Refl
TBool =?= TBool           = Just Refl
-- New -----------------------------------------
(TList t) =?= (TList t') = do Refl <- t =?= t'
                              return Refl
_     =?= _     = Nothing

infer :: Ex1.Expr -> Maybe TypedExpr
infer e = case e of
  Ex1.LitI n -> return (LitI n ::: TInt)

  Ex1.LitB b -> return (LitB b ::: TBool)

  r1 Ex1.:+: r2 -> do
    e1 ::: t1  <-  infer r1
    e2 ::: t2  <-  infer r2
    Refl       <- t1 =?= t2
    case t1  of
      TInt       -> return (e1 :+: e2 ::: TInt)
      TList TInt -> return (e1 :+: e2 ::: TList TInt)
      _          -> Nothing

  r1 Ex1.:==: r2 -> do
    e1 ::: t1    <-  infer r1
    e2 ::: t2    <-  infer r2
    Refl         <-  t1 =?= t2
    return (e1 :==: e2 ::: TBool)
```

```
  Ex1.If r1 r2 r3 -> do
    e1 ::: TBool <-  infer r1
    e2 ::: t2    <-  infer r2
    e3 ::: t3    <-  infer r3
    Refl         <-  t2 =?= t3
    return (If e1 e2 e3 ::: t2)

  Ex1.Cons x Ex1.Nil -> do
    x1 ::: tx <- infer x
    return (Cons x1 Nil ::: TList tx)

  Ex1.Cons x xs -> do
    x1 ::: tx  <- infer x
    xs ::: txs <- infer xs
    Refl       <- txs =?= TList tx
    return (Cons x1 xs ::: txs)

  Ex1.Nil -> Nothing

check :: Ex1.Expr -> Type t -> Maybe (Expr t)
check r t = do
  e ::: t' <- infer r
  Refl     <- t' =?= t
  return e

-- New -----------------------------------------
evalT :: Ex1.Expr -> Maybe Ex1.Value
evalT e = do
  te ::: typee <- infer e
  -- Key idea: to evaluate everything and then injects tags
  return $ buildValue typee (eval te)
  where
    buildValue :: Type t -> t -> Ex1.Value
    buildValue TBool      b    = Ex1.VBool b
    buildValue TInt       i    = Ex1.VInt  i
    buildValue (TList tx) ls   = Ex1.VList $ map (buildValue tx) ls

prop_eval :: Ex1.Expr -> Property
prop_eval e = let  mv          = evalT e
                   wellTyped  = isJust   mv
                   v          = fromJust mv
              in wellTyped ==>
                 label (Ex1.showTypeOfVal v) $
                 Ex1.eval e == v

-- | Check that the evals agree for well-typed terms
main :: IO ()
main = quickCheck prop_eval

ok1  = Cons (LitI 1) (Cons (LitI 2) Nil)
--bad1 = Cons (LitI 1) (Cons (LitB True) Nil)
--bad2  = Cons (Cons (LitI 1) Nil) (Cons (LitI 2) Nil)
ok2 = Cons (Cons (LitI 1) Nil) (Cons (Cons (LitI 2) Nil) Nil)
ok3 = ok1 :+: ok1
ok4 = Cons (LitB True) Nil
ok5 = ok4 :==: ok4
--bad3 = ok4 :==: ok3
```