```
-- ****************************************************************************
-- *                                                                          *
-- *                  Less Trivial eXpression Language (LTXL)                 *
-- *                                                                          *
-- *    Module:         AbstractSyntax                                        *
-- *    Purpose:        Definition of abstract syntax for LTXL                *
-- *    Author:         Henrik Nilsson                                        *
-- *                                                                          *
-- *           Example for G53CMP, lecture 10, November 2017                  *
-- *                                                                          *
-- ****************************************************************************

module AbstractSyntax (
    Id,
    UnOp(..),
    BinOp(..),
    Exp(..),
    ppExp -- :: Exp -> String
) where

import Type

type Id = String

data UnOp = Not | Neg deriving (Eq, Show)

data BinOp = Or
           | And
           | Less
           | Equal
           | Greater
           | Plus
           | Minus
           | Times
           | Divide
           deriving (Eq, Show)

data Exp = LitInt   Int
         | Var      Id
         | UnOpApp  UnOp Exp
         | BinOpApp BinOp Exp Exp
         | If       Exp Exp Exp
         | Let      [(Id, Type, Exp)] Exp
         deriving Show


------------------------------------------------------------------------------
-- Pretty printing of expressions
------------------------------------------------------------------------------

[OMITTED]
```



```
-- ****************************************************************************
-- *                                                                          *
-- *                  Less Trivial eXpression Language (LTXL)                 *
-- *                                                                          *
-- *    Module:         Environment                                           *
-- *    Purpose:        Type-checking environment and operations              *
-- *                    Adapted to work with Diagnostics from HMTC            *
-- *    Author:         Henrik Nilsson                                        *
-- *                                                                          *
-- *           Example for G53CMP, lecture 10, November 2017                  *
-- *                                                                          *
-- ****************************************************************************

module Environment (
    VarAttr,
    Env,        -- Abstract
    glblLvl,    -- :: Int
    initEnv,    -- :: [(Id, Type)] -> [(UnOp, Type)] -> [(BinOp, Type)] -> Env
    enterVar,   -- :: Id -> Int -> Type -> Env -> Either Env String
    lookupVar,  -- :: Id -> Env -> Either VarAttr String
    lookupUO,   -- :: UnOp -> Env -> Type
    lookupBO    -- :: BinOp -> Env -> Type
) where

import Data.List (nub)

import Diagnostics      -- Refers to the Diagnostics module from HMTC
import AbstractSyntax (Id, UnOp, BinOp)
import Type (Type)


------------------------------------------------------------------------------
-- Environment (Symbol table)
------------------------------------------------------------------------------

-- The environment stores information about entities, in particular their
-- type.
--
-- There are three kinds of entities. Each have a distinct kind of name
-- (used for looking it up), and a distinct set of attributes:
--   * Variables
--       Name: Identifier
--       Attributes: Scope level and type.
--   * Unary operator
--       Name: UnOp
--       Attributes: Type only (the scope level is always the global one).
--   * Binary operator
--       Name: BinOp
--       Attributes: Type only (the scope level is always the global one).
--
--
-- The reason that operators are separated from variables is that the LTXL
-- syntax defines a fixed set of operators (user-defined operators are not
-- supported). For the same reason, we also know that an operator is an entity
-- defined at the global level.
--
-- The environment is an Abstract Data Type (ADT) which is achieved by not
-- exporting its constructor from this module. It is split into three parts,
-- one for each kind of entity. For the sake of simplicity, we use a linear
-- association list to represent the environment. By prepending new
-- declarations to the list, and searching from the beginning, it is ensured
-- that we will always find an identifier in the closest containing scope, as
-- specified by the LTXL Scope Rules.
```

```
type VarAttr = (Int, Type)

data Env = Env {
               varEnv :: [(Id, VarAttr)],       -- The variable environment
               uoEnv  :: [(UnOp, Type)],        -- The unary operator env.
               boEnv  :: [(BinOp, Type)]        -- Tye binary operator env.
          }


glblLvl :: Int
glblLvl = 0


-- Creates an initial environment (everything defined at the top level).

initEnv :: [(Id, Type)] -> [(UnOp, Type)] -> [(BinOp, Type)] -> Env
initEnv ve uoe boe =
    if ok then
        Env {
            varEnv = [ (i, (glblLvl, tp)) | (i, tp) <- ve ],
            uoEnv  = uoe,
            boEnv  = boe
        }
    else
        internalError "Environmant" "initEnv"
                      "Bad initial environment: multiply defined entities"
    where
        ok = length is == length (nub is)
             && length uos == length (nub uos)
             && length bos == length (nub bos)
        is  = map fst ve
        uos = map fst uoe
        bos = map fst boe


-- Enters a variable at the given scope level and of the given type into the
-- environment. A check is first performed so that the no other variable with
-- the same name has been defined at the same scope level. If not, the new
-- variable is entered into the table. Otherwise an error message is returned
-- (string).

enterVar :: Id -> Int -> Type -> Env -> Either Env String
enterVar i l t env@(Env {varEnv = ve})
    | not (isDefined i l ve) = Left (env {varEnv = ((i,(l,t)) : ve)})
    | otherwise              = Right errMsg
    where
        isDefined i l [] = False
        isDefined i l ((i',(l',_)) : ve)
            | l < l'  = error "Should not happen! (Bad level.)"
            | l > l'  = False
            | i == i' = True
            | otherwise = isDefined i l ve

        errMsg = "Variable " ++ i
                 ++ " is already defined at this level ("
                 ++ show l ++ ")."
```



```
-- Looks up a variable and returns its attributes if found.
-- Otherwise returns an error message (string).

lookupVar :: Id -> Env -> Either VarAttr String
lookupVar i env = lvRec i (varEnv env)
    where
        lvRec i [] = Right ("Variable " ++ i ++ " is not defined.")
        lvRec i ((i',a) : ve)
            | i == i'   = Left a
            | otherwise = lvRec i ve


-- Looks up a unary operator and returns its type.
-- It should always be found, otherwise there is an error in the compiler.

lookupUO :: UnOp -> Env -> Type
lookupUO uo env =
    case lookup uo (uoEnv env) of
        Just tp -> tp
        Nothing -> internalError "Environment" "lookupUO"
                                 ("Unknown unary operator " ++ show uo)


-- Looks up a binary operator and returns its type.
-- It should always be found, otherwise there is an error in the compiler.

lookupBO :: BinOp -> Env -> Type
lookupBO bo env =
    case lookup bo (boEnv env) of
        Just tp -> tp
        Nothing -> internalError "Environment" "lookupBO"
                                 ("Unknown unary operator " ++ show bo)
```

```
-- ****************************************************************************
-- *                                                                          *
-- *                  Less Trivial eXpression Language (LTXL)                 *
-- *                                                                          *
-- *    Module:         GlobalEnvironment                                     *
-- *    Purpose:        The global environment                                *
-- *    Author:         Henrik Nilsson                                        *
-- *                                                                          *
-- *           Example for G53CMP, lectures 10, November 2017                 *
-- *                                                                          *
-- ****************************************************************************

module GlobalEnvironment where

import AbstractSyntax (Id, UnOp(..), BinOp(..))
import Type (Type(..))
import Environment (Env, initEnv)


glblEnv :: Env
glblEnv =
    initEnv []                          -- No globally defined variables
            [ (Not, tpBtoB),            -- The unary operators
              (Neg, tpItoI) ]
            [ (Or,      tpBBtoB),       -- The binary operators
              (And,     tpBBtoB),
              (Less,    tpIItoB),
              (Equal,   tpIItoB),
              (Greater, tpIItoB),
              (Plus,    tpIItoI),
              (Minus,   tpIItoI),
              (Times,   tpIItoI),
              (Divide,  tpIItoI) ]
    where
        tpBtoB  = TpArr TpBool TpBool
        tpItoI  = TpArr TpInt TpInt
        tpBBtoB = TpArr (TpProd TpBool TpBool) TpBool
        tpIItoB = TpArr (TpProd TpInt TpInt) TpBool
        tpIItoI = TpArr (TpProd TpInt TpInt) TpInt
```



```
-- ****************************************************************************
-- *                                                                          *
-- *                  Less Trivial eXpression Language (LTXL)                 *
-- *                                                                          *
-- *    Module:         TypeChecker                                           *
-- *    Purpose:        Type checker for LTXL using diagnostics monad         *
-- *                    from HMTC                                             *
-- *    Author:         Henrik Nilsson                                        *
-- *                                                                          *
-- *           Example for G53CMP, lectures 10, November 2017                 *
-- *                                                                          *
-- ****************************************************************************

module TypeChecker (typeCheck, tcAux) where

import Control.Monad (unless)

import SrcPos           -- Module from HMTC
import Diagnostics      -- Refers to the Diagnostics module from HMTC
import AbstractSyntax
import Type
import Environment
import GlobalEnvironment


------------------------------------------------------------------------------
-- Type Checker
------------------------------------------------------------------------------

-- The type checker checks whether LTXL terms are well-typed. In the process,
-- it also does "identification" as it has to link up definitions and uses
-- of variables.
--
-- A "real" type checker might return an AST annotated with type information
-- and information about the scope level of variables etc., which then is used
-- for optimization and code generation. However, to keep things simple, this
-- type checker just checks types and returns the type of a term (if any)
-- along with a list of identification and type errors.
--
-- Important: A term is well-typed if and only if no error messages are
-- returned. Only then can the returned type be "trusted".
--
-- The "type" TpUnknown is returned (along with an error message) when there
-- is a problem and there is no "better" type to return. To avoid one type
-- error causing lots of other type errors, the type "TpUnknown" is treated as
-- "compatible" with *any* other type. This could potentially mask some
-- further real errors, but is preferable to reporting lots of problems when
-- there is only one error.
--
-- Written for clarity, not efficiency.

typeCheck :: Exp -> (Maybe Type, [DMsg])
typeCheck e = runD (tcAux 0 glblEnv e)

tcAux :: Int -> Env -> Exp -> D Type
tcAux l env (LitInt n) = undefined
tcAux l env (Var i)    = undefined
tcAux l env (UnOpApp uo e1) = do
    let TpArr t1 t2 = lookupUO uo env                   -- Safe if comp. corr.
    t1' <- tcAux l env e1
    unless (compatible t1 t1') (emitErrD NoSrcPos (illTypedOpApp t1 t1'))
    return t2
tcAux l env (BinOpApp bo e1 e2) = undefined
tcAux l env (If e1 e2 e3)       = undefined
tcAux l env (Let ds e)          = undefined
```

```
-- For simplified LTXL scope rules (a defined variable NOT in scope in
-- further definitions of the same let).
tcDefs :: Int -> Env -> [(Id, Type, Exp)] -> D Env
tcDefs l env [] = return env
tcDefs l env ((i, t, e) : ds) = do
    t'   <- tcAux l env e                              -- i NOT in scope!
    env' <- tcDefs l env ds                            -- i NOT in scope!
    unless (compatible t t') (emitErrD NoSrcPos (declMismatch t t'))
    case enterVar i l t env' of
        Left env'' -> return env''
        Right m    -> emitErrD NoSrcPos m >> return env'


--------------------------------------------------------------------------------
-- Utilities
--------------------------------------------------------------------------------

-- Type comparison where "TpUnknown" is treated as compatible with any
-- other type to avoid one type error causing further errors.

compatible :: Type -> Type -> Bool
compatible TpUnknown _         = True
compatible _         TpUnknown = True
compatible t1        t2        = t1 == t2


illTypedOpApp :: Type -> Type -> String
illTypedOpApp t1 t2 =
    "Ill-typed operator application: expected type " ++ ppType t1
    ++ ", got type " ++ ppType t2


illTypedCond :: Type -> String
illTypedCond t = "Ill-typed condition: expected bool, got " ++ ppType t


incompatibleBranches :: Type -> Type -> String
incompatibleBranches t1 t2 =
    "Expected same type in both then and else branch, but got types "
    ++ ppType t1 ++ " and " ++ ppType t2


declMismatch :: Type -> Type -> String
declMismatch t1 t2 =
    "Declared type " ++ ppType t1
    ++ " does not macth inferred type " ++ ppType t2
```



```
-- ****************************************************************************
-- *                                                                          *
-- *                  Less Trivial eXpression Language (LTXL)                 *
-- *                                                                          *
-- *    Module:         Type                                                  *
-- *    Purpose:        Representation of types                               *
-- *    Author:         Henrik Nilsson                                        *
-- *                                                                          *
-- *           Example for G53CMP, lectures 10, November 2017                 *
-- *                                                                          *
-- ****************************************************************************

module Type (
    Type(..),
    ppType -- :: Type -> String
) where

data Type = TpUnknown           -- "Type" of ill-typed terms
          | TpBool              -- Boolean type
          | TpInt               -- Integer type
          | TpProd Type Type    -- Product type ("pair", (T1,T2))
          | TpArr  Type Type    -- Function type (T1 -> T2)
          deriving (Eq, Show)


--------------------------------------------------------------------------------
-- Pretty printing of types
--------------------------------------------------------------------------------

ppType :: Type -> String
ppType TpUnknown      = "???"
ppType TpBool         = "bool"
ppType TpInt          = "int"
ppType (TpProd t1 t2) = "(" ++ ppType t1 ++ "," ++ ppType t2 ++ ")"
ppType (TpArr t1 t2)  = "(" ++ ppType t1 ++ "->" ++ ppType t2 ++ ")"
```