```haskell
module MikeIRGen where

import AST
import qualified SymbolTable as S
import IRDataType
import ST
import qualified Semantic as SEM

transProgIR :: AST -> I_prog
transProgIR (M_prog(mdec,mstmt)) = IPROG (fcnList, len , stmts) where
    fcnList' = (filter(\x -> not (isVar x))) mdec
    len = length (filter isVar(mdec))
    st = S.beginProcess (M_prog(mdec,mstmt))
    fcnList = transMdecls st fcnList'
    stmts = transMstmts st mstmt
   {- fcnList = case semanticResult of
        True -> transMdecls st fcnList'
    stmts = case semanticResult of
        True -> transMstmts st mstmt
    semanticResult = case typeProg st (M_prog(mdec,mstmt)) of
        True -> True
        False -> error ("Semantic Analysis Produced an Error")-}



transMdecls :: ST -> [M_decl] -> [I_fbody]
transMdecls st [] = []
transMdecls st (x:xs) = case x of
    M_fun x -> case (SEM.checkDecls st ((M_fun x):xs)) of
        True -> (transMdecl st (M_fun x)):(transMdecls st xs)
    x -> exp where -- to catch and show the error
        exp = error("error " ++ show(exp))

transMdecl :: ST -> M_decl -> I_fbody
transMdecl st (M_fun(name,triple,rTyp,dec,stm)) = IFUN(name,iFcns,numVars,numArgs,stmts) where
        numArgs = (length(filter(\(s,n,t) -> n < 0) triple))
        numVars = (length triple) - numArgs
        iFcns' = (filter(\x -> not (isVar x))) dec
        iFcns = map(\x -> convertMfun st x) iFcns'
        stmts = case SEM.checkStmts st stm of
            True -> transMstmts st stm
            False -> error("irgen line 41: ")
transMstmts :: ST -> [M_stmt] -> [I_stmt]
transMstmts st [] = []
transMstmts st (x:xs) = case (SEM.checkStmts  st (x:xs)) of
    True -> (transMstmt st x):(transMstmts st xs)
    False -> error ("irgen line 46: ")



transMstmt :: ST -> M_stmt -> I_stmt
transMstmt st x = case x of
    M_ass (str,expList,exp) -> testing where
         expr = (convertMexpr st exp) -- some problem here??
         testing = case (S.look_up st str) of
            I_VARIABLE(lev,off,_,_) -> IASS(lev,off,expr)
            x -> error("MikeIRGen line: 57 transMstmt: " ++ show(x))
    M_while (e,s) -> IWHILE(exp,stm) where
        exp = convertMexpr st e
        stm = transMstmt st s
    M_cond (e,s1,s2) -> ICOND(exp,stm1,stm2) where
        exp = convertMexpr st e
        stm1 = transMstmt st s1
        stm2 = transMstmt st s2
    M read (s,exp) -> case  (S.look up st s) of
        I VARIABLE(lev,off,M bool, ) -> IREAD B(lev,off)
        I VARIABLE(lev,off,M int, ) -> IREAD I(lev,off)
    M print (M bval exp) -> IPRINT B (IBOOL exp)
    --M print (M ival exp) -> IPRINT I (INUM (fromInteger exp))
    M print e -> IPRINT I exp where
        exp = convertMexpr st e
    M return e -> IRETURN exp where
        exp = convertMexpr st e
    M block (dec,stm) -> IBLOCK (fbdys,locV,stmts) where
        fbdys' = (filter(\x -> not (isVar x))) dec
        fbdys = transMdecls st fbdys'
        locV = length(dec) - length(fbdys)
        stmts = transMstmts st stm



convertMexpr :: ST -> M expr -> I expr
convertMexpr st x = case x of
    M ival y -> (INUM (fromInteger(y)))
    M bval y -> (IBOOL y)
    M id (str, ) -> IID(lev,off) where
        I VARIABLE(lev,off, , ) = S.look up st str
    M app (mOP,exp) -> IAPP (opn, expI) where
        opn = convertMop st mOP
        expI = map(\x -> convertMexpr st x) exp


convertMop :: ST -> M operation -> I opn
convertMop  st x = case x of
    M fn (str) -> ICALL(lbl,lev) where
        I FUNCTION(lev,lbl, , ) = S.look up st str
    M add -> IADD
    M mul -> IMUL
    M sub -> ISUB
    M div -> IDIV
    M neg -> INEG
    M lt  -> ILT
    M le  -> ILE
    M gt  -> IGT
    M ge  -> IGE
    M eq  -> IEQ
    M not -> INOT
    M and -> IAND
    M or  -> IOR
   -- M float
    --M floor
    --M ceil



isVar :: M decl -> Bool
isVar m = case m of
    M var m -> True
      -> False

convertMfun :: ST -> M decl -> I fbody
convertMfun st (M fun (name,triple,typ,decls,stmts)) = IFUN
(name,fcnList,locV,locA,istmts) where
    Symbol table( ,locV,locA, ) = (st !! lev) where
        I FUNCTION(lev, , , ) = S.look up st name
    fcnList = case (SEM.checkDecls st decls) of
        True -> transMdecls st decls
        False -> error("IrGen line 127: ")
    istmts = case (SEM.checkStmts st stmts) of
```

```haskell
          True -> transMstmts st stmts
          False -> error("irGen line 130")

```