```
module Types where
import qualified Data.Set as Set
import qualified Data.Map as Map

data PatToken = PTWild | PTConstr String
  deriving (Show, Eq, Ord)

data Pattern = Pat PatToken (Maybe Pattern)
  deriving (Show, Eq, Ord)

type Env = Map.Map PatToken (Map.Map PatToken Int)

patwild :: Pattern
patwild = Pat PTWild Nothing

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

data Pat =
  PatLit Lit
  | PatObj Constr (Maybe [Pat])
  | PatTuple [Pat]
  | PatWild
  deriving (Show, Eq, Ord)

data Lit = S String | I Int | C Char | B Bool
  deriving (Show, Eq, Ord)

data Constr = Constr String
  deriving (Show, Eq, Ord)

data Match = Match String [Pat]
  deriving (Show, Eq, Ord)

convertLit :: Lit -> PatToken
convertLit x = case x of
   (S s) -> PTConstr s
   (I i) -> PTConstr $ show i
   (C c) -> PTConstr $ show c
   (B b) -> PTConstr $ show b

convertConstr :: Constr -> PatToken
convertConstr (Constr s) = PTConstr s

convertPat :: Pat -> Maybe Pattern -> Pattern
convertPat (PatLit l) k = Pat (convertLit l) k
convertPat PatWild k = Pat PTWild k
convertPat (PatTuple [p]) k = Pat (PTConstr "tuple") $ Just $ convertPat p k
convertPat (PatTuple (p:ps)) k =
  let rest = Just $ convertPat (PatTuple ps) k in
  let pk = convertPat p rest in
  Pat (PTConstr "tuple") $ Just pk
convertPat (PatObj c Nothing) k = Pat (convertConstr c) k
convertPat (PatObj c (Just pats)) k =
  let rest = convertPat (PatTuple pats) k in
  Pat (convertConstr c) $ Just rest

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
env1 :: Map.Map PatToken Int
env1 = Map.fromList [(PTConstr "True", 0), (PTConstr "False", 0)]
env2 = Map.fromList [(PTConstr "True", env1), (PTConstr "False", env1)]

t = Match "f" [PatLit (B True), PatLit (B False)]
tb = Match "f" [PatLit (B True), PatLit (B True)]
ttb = Match "f" [PatLit (B True), PatWild]
ttf = Match "f" [PatLit (B True)]

tt = Match "f" [PatWild]

ps1 = [(PatObj (Constr "Nil") Nothing), (PatObj (Constr "Cons") (Just [PatLit (I 1),
 PatWild]))]
```

```
e1 :: Map.Map PatToken Int
e1 = Map.fromList [(PTConstr "Nil", 0), (PTConstr "Cons", 2)]
e2 = Map.fromList [(PTConstr "Nil", e1), (PTConstr "Cons", e1)]
t3 = Match "f" [(PatObj (Constr "Nil") Nothing)]
t4 = Match "f" [(PatObj (Constr "Nil") Nothing), (PatObj (Constr "Cons") (Just [PatW
ild, PatWild]))]
t5 = Match "f" [(PatObj (Constr "Nil") Nothing), (PatObj (Constr "Cons") (Just [PatL
it (I 1), PatWild]))]

p5 = [(PatObj (Constr "Nil") Nothing), (PatObj (Constr "Cons") (Just [PatLit (I 1),
PatWild])), PatWild]
p5' = [(PatObj (Constr "Nil") Nothing), (PatObj (Constr "Cons") (Just [PatLit (I 1),
 PatWild]))]
p6 = [(PatObj (Constr "Nil") Nothing), (PatObj (Constr "Cons") (Just p5'))]
p6' = [(PatObj (Constr "Nil") Nothing), (PatObj (Constr "Cons") (Just p5')), PatWild
]
t6 = Match "f" p5
t7nonex = Match "f" p6
t7ex = Match "f" p6'

p1 = PatObj (Constr "Cons") (Just [PatLit (I 1), PatWild])

{-
Cons 1 Nil

Nil -> Pat "Nil" Nothing
1 -> Pat "1" Nothing
1 Nil -> Pat "1" (Just (Pat "Nil" Nothing))
(1, Nil) -> Pat "tuple" (Just (Pat "1" (Just (Pat "Nil" Nothing))))
Cons(1, Nil) -> Pat "Cons" (Just (Pat "tuple" (Just (Pat "1" (Just (Pat "Nil" Nothin
g))))))
-}
```