PrettyPrinter.hs

```haskell
module PrettyPrinter (
       printTerm,     -- pretty printer para terminos
       printType,     -- pretty printer para tipos
       )
       where

import Common
import Text.PrettyPrint.HughesPJ

-- lista de posibles nombres para variables
vars :: [String]
vars = [ c : n | n <- "" : map show [1..], c <- ['x','y','z'] ++ ['a'..'w'] ]

parensIf :: Bool -> Doc -> Doc
parensIf True  = parens
parensIf False = id

-- pretty-printer de términos
-- r APP > r SND FST > r SUC > r REC > l AS > r Lam Let > r -> > l =
--( l * \ > l + -)

pp :: Int -> [String] -> Term -> Doc
pp ii vs (Bound k)         = text (vs !! (ii - k - 1))
pp _  vs (Free (Global s)) = text s
pp ii vs (i :@: c) = sep [parensIf (isMenorApp i) (pp ii vs i),
                      nest 1 (parensIf  (isNotAtom c ) (pp ii vs c))]
pp ii vs (Lam t c) = text "\\" <>
                  text (vs !! ii) <>
                  text ":" <>
                  printType t <>
                  text ". " <>
                  pp (ii+1) vs c
pp ii vs (Let t u) = text "let " <>
                  text (vs !! ii) <>
                  text " = " <>
                  (pp ii vs t) <>
                  text " in " <>
                  pp (ii+1) vs u
pp ii vs (As t tt) = sep [parensIf (isMenorAs t ) (pp ii vs t),
                  (text "as"),
                  (printType tt)]
pp ii vs Unit      = text "unit"
pp ii vs (Pair t1 t2) = parens $
                   pp ii vs t1 <> text "," <> pp ii vs t2
pp ii vs (Fst t) = text "fst " <> parensIf (isNotAtom t) (pp ii vs t)

pp ii vs (Snd t) = text "snd " <> parensIf (isNotAtom t) (pp ii vs t)

pp ii vs Zero = text "0"
pp ii vs (Suc t) = text "suc " <> parensIf (isNotAtom t) (pp ii vs t)
pp ii vs (R t1 t2 t3) = sep[ text "R ",
                             parensIf (isNotAtom  t1) (pp ii vs t1),
                             parensIf (isNotAtom  t2) (pp ii vs t2),
                             parensIf (isApp t3 || isAs t3) (pp ii vs t3)
                             ]

isMenorAs  i = isLam i || isLet i
isMenorRec i = isAs i  || isMenorAs i
isMenorSuc i = isRec i || isMenorRec i
isMenorFst i = isSuc i || isMenorSuc i
isMenorApp i = isMenorFst i || isFst i || isSnd i
isNotAtom  i = isMenorApp i || isApp i

isRec (R _ _ _) = True
isRec _         = False

isSuc (Suc _) = True
isSuc _       = False

isFst (Fst _) = True
isFst _ = False

isSnd (Snd _) = True
isSnd _ = False


isAs (As _ _ ) = True
isAs _         = False

isLet (Let _ _) = True
isLet _          = False

isLam (Lam _ _) = True
isLam  _      = False

isApp (_ :@: _) = True
isApp _         = False

-- pretty-printer de tipos
printType :: Type -> Doc
printType Base         = text "B"
printType (Fun t1 t2)  = sep [ parensIf (isFun t1) (printType t1),
                         text "->",
                         printType t2]
printType TUnit        = text "Unit"
printType (TPar t1 t2) = parens $
                         printType t1 <>
                         text "," <>
                         printType t2
printType TNat         = text "Nat"
```

```
isFun (Fun _ _)        = True
isFun _                = False

fv :: Term -> [String]
fv (Bound _)         = []
fv (Free (Global n)) = [n]
fv (Free _)          = []
fv (t :@: u)         = fv t ++ fv u
fv (Lam _ u)         = fv u
fv (Let t u)         = fv t ++ fv u
fv (As t tt)         = fv t
fv (Unit)            = []
fv (Pair t u)        = fv t ++ fv u
fv (Fst t)           = fv t
fv (Snd t)           = fv t
fv (Zero)            = []
fv (Suc t)           = fv t
fv (R t1 t2 t3)      = fv t1 ++ fv t2 ++ fv t3

---
printTerm :: Term -> Doc
printTerm t = pp 0 (filter (\v -> not $ elem v (fv t)) vars) t
```