```haskell
import Text.ParserCombinators.Parsec

infix 6 :-
type Prop = String
data Lit = PL Prop | NL Prop  deriving (Show, Eq)
data TAtom = U Lit Prop | R Lit Prop | F Prop | G Prop  deriving (Show, Eq)
data Atom = A Prop | TA TAtom | X Atom  deriving (Show, Eq)
data NowClause  = [Atom] :- [Atom]  deriving (Show, Eq)
data Clause  = Now NowClause | Alw NowClause  deriving (Show, Eq)

type Program = [Clause]

{-
-- EJEMPLO (más abajo otra vez)
G [U waitingDV ackSM] :- [reqDV].
G [X F ackSM] :- [reqDV].
G [U workingDV eopDV] :- [ackSM].
G [U Not workingDV ackSM] :- [ackSM].
G [F ctrSM].
G [F connSM] :- [ctrSM].
G [X F connDV] :- [connSM].
G [comDV] :- [F connDV].
-}

---- Parser ----

{---------------- SOURCE LANGUAGE -----------------------------
Prop = String
    -- starting for any lowercase letter except 'n', which is reserved for new variables
Lit = Prop | Not Prop
TAtom = U Lit Prop | R Lit Prop | F Prop | G Prop
Atom = Prop | TAtom | X Atom
NowClause  = [Atom] :- [Atom] | [Atom].
    -- el segundo tipo de NowClause es un caso particular donde la segunda lista es vacía.
Clause  = NowClause. | G NowClause.
-------------------------------------------------------------}

-- Spaces are always consumed with the previous token.
-- Los espacios son leidos siempre trás el símbolo anterior

schar :: Char -> GenParser Char a ()
schar c = char c >> spaces

sstring :: [Char] -> GenParser Char a ()
sstring s = string s >> spaces

-- Un parser para cada entidad sintáctica del lenguaje fuente.

prop :: GenParser Char a [Char]
prop = (oneOf "abcdefghijklmopqrstuvwxyz" >>= \x -> many alphaNum >>= \xs -> spaces >> return
  (x:xs))
-- parse prop "" "atom1234n"
-- parse prop "" "n1234"

literal :: GenParser Char a Lit
literal = (prop >>= \prop -> spaces >> return (PL prop)) <|>
          (sstring "Not" >>  prop >>= \prop -> spaces >> return (NL prop))
-- parse literal "" "Not p1234n"
```

```
-- parse literal "" "Not n1234n"

tatom :: GenParser Char a TAtom
tatom = (schar 'U' >> literal >>= \lit -> prop >>= \prop -> spaces >> return(U lit prop)) <|>
        (schar 'R' >> literal >>= \lit -> prop >>= \prop -> spaces >> return(R lit prop)) <|>
        (schar 'F' >> prop >>= \prop -> spaces >> return(F prop)) <|>
        (schar 'G' >> prop >>= \prop -> spaces >> return(F prop))
-- parse tatom "" "U Not p1 q2"
-- parse tatom "" "U p11 q22"
-- parse tatom "" "U n11 q22"

atom :: GenParser Char a Atom
atom = (prop >>= \prop -> spaces >> return (A prop)) <|>
       (tatom >>= \tatom -> spaces >> return (TA tatom)) <|>
       (sstring "X" >> atom >>= \atom -> spaces >> return (X atom))
-- parse atom "" "X X X  q123 "
-- parse atom "" "X X X  U Not q123 p3"

listatoms :: GenParser Char a [Atom]
listatoms = schar '[' >> atoms >>= \aas -> schar ']' >> return aas
            where
            atoms :: GenParser Char a [Atom]
            atoms = sepBy atom (schar ',')

query :: GenParser Char a [Atom]
query = arrow >> listatoms >>= \body -> schar '.' >> return body
        where
        arrow :: GenParser Char a ()
        arrow = (char '?' <|> char ':') >> schar '-'

nowclause :: GenParser Char a NowClause
nowclause = listatoms >>= \head -> ((schar '.' >> return (head :- [])) <|>
                                (query >>= \body -> return (head :- body)))

clause :: GenParser Char a Clause
clause = (nowclause >>= \nc -> return (Now nc)) <|>
         (schar 'G' >> nowclause >>= \nc -> return (Alw nc))
-- parse clause "" "G [F  r, G b, R Not p q] :- [U Not p q, G a] . "
-- parse clause "" "G [F  r, G b, R p Not q] :- [U Not p q, G a] . "
-- parse clause "" "G [F  r, G n, R Not p q] :- [U Not p q, G a] . "
-- parse clause "" "[F  r, G c, R Not p q] :- [U Not p q, G Not a , F v]."
-- parse clause "" "[F  r, G c, R Not p q] :- [U Not p q, G a , F v]."
-- parse clause "" "G [F ctrSM]."

program :: GenParser Char a [Clause]
program = many clause
-- parse program "" "G [F  r, G b, R Not p q] :- [U Not p q, G a] .  [F  b]. [R p1 q2] :- [
G b34 , F b67]."

miParser :: GenParser a () b -> [a] -> b
miParser parser s = result
                    where Right result = parse parser "" s

-- Un ejemplo de programa tal que todos sus modelos cumplen G comDV
c1 = "G [U waitingDV ackSM] :- [reqDV]."
c2 = "G [X F ackSM] :- [reqDV]. "
c3 = "G [U workingDV eopDV] :- [ackSM]."
c4 = "G [U Not workingDV ackSM] :- [ackSM]."
```

```
c5 = "G [F ctrSM]."
c6 = "G [F connSM] :- [ctrSM]."
c7 = "G [X F connDV] :- [connSM]."
c8 = "G [comDV] :- [F connDV]."

system = miParser program (c1++c2++c3++c4++c5++c6++c7++c8)
-- system = parse program "" (c1++c2++c3++c4++c5++c6++c7++c8)

-- Interface
main:: IO ()
main =  do
        putStrLn "Nombre/Path del fichero que contiene el programa: "
        pathPrograma <- getLine
        prog <- leerPrograma pathPrograma
        putStrLn (show prog)
        -- putStrLn "Nombre/Path del fichero que contiene el modelo: "
        -- pathModelo <- getLine
        -- model <- leerModelo pathModelo       -- leerModelo debes definirla
        -- devolver el resultado de comprobar si model satisface prog

leerPrograma :: [Char] -> IO [Clause]
leerPrograma path = do
                    result <- parseFromFile program path
                    case result of
                          Left _ -> return (error "programa sintacticamente incorrecto")
                          Right prog -> return prog

{- FASE 1:
Hacer que los programas se muestren en el lenguaje fuente (ocultar la representación
interna), pero sin los blancos adicionales y con un salto de linea tras cada clausula. Para
ello:
instance Show Lit where ...
instance Show TAtom where ...
instance Show Atom where ...
instance Show NowClause where ...
instance Show Clause where ..
Definir una función showProgram y cambiar en main "show prog" por "showProgram prog".
-}

{- FASE 2:
Esta es la fase principal del trabajo y la que debe llevar la mayoría del tiempo de
programación. Para realizarla debe seguirse la documentación dada en el fichero
TeDiLog-Model-Checking.pdf y consiste en:
    - Definir un tipo de datos para representar modelos.
    - Implementar una función que decida si un modelo satisface un programa.
-}

{- FASE 3:
- Implementar la función leerModelo.
- Adaptar main para que se lea el modelo y se muestren los resultados.
-}

{- FASE 4 (opcional):
Refactorizar el programa para que en caso de que model no satisfaga la fórmula nos informe
de cúal es la (primera) cláusula que no se satisface.
-}
```