First, we want some imports:

1 Parser

```
module Main where
import Text.Megaparsec hiding (State)
import Text.Megaparsec.Char
import qualified Text.Megaparsec.Char.Lexer as L
import qualified Data.Text as T
import qualified Data.Text.IO as T
import Data.Void
import Data.Foldable
import qualified Data.Map as Map
import qualified Data.Set as Set
import Control.Monad.State
import Control.Monad (void)
```

Then, we add some boilerplate:

```
lexeme :: Parser a → Parser a
lexeme = L.lexeme space

symbol :: T.Text → Parser T.Text
symbol = L.symbol space

packSymbol :: String → Parser T.Text
packSymbol = symbol ∘ T.pack

parens :: Parser a → Parser a
parens = between (packSymbol "(") (packSymbol ")")
```

We rewrite the following grammar into haskell:

```
<formula> ::= '(' 'and' <formula> <formula> ')'
            | '(' 'or' <formula> <formula> ')'
            | '(' 'not' <variable> ')'
            | <variable>
```

```
formula :: Parser Formula
formula = try (parens $ do packSymbol "and" ≫ FAnd <$> formula <*> formula
                         <|> do packSymbol "or" ≫ FOr <$> formula <*> formula
                         <|> do packSymbol "not" ≫ FNeg ∘ FVar <$> variable
                        )
                         <|> do FVar <$> variable
variable :: Parser Variable
variable = lexeme $ (pure <$> letterChar) <> many alphaNumChar
```

And then we define the supporting types:

```
type Parser = Parsec Void T.Text
data Formula = FAnd Formula Formula
  | FOr Formula Formula
  | FNeg Formula
  | FVar Variable
  deriving (Eq, Ord, Show)
type Variable = String
```

2 Encoding

Now we can define the encoding:

```
encode f = do
  name ← encode' f
  modify (λstate → state { cnfRepr = [name] `Set.insert` cnfRepr state })
encode' :: Formula → Encoder Int
encode' f@(FAnd left right) = do
  leftName ← encode' left
  rightName ← encode' right
  name ← getName f
  state@EncoderState { cnfRepr = cnf } ← get
  put state { cnfRepr = foldr Set.insert cnf [[-name, leftName], [-name, rightName], [-leftName, -
  return name
encode' (FOr left right) = encode' ∘ FNeg $ FAnd (FNeg left) (FNeg right)
encode' (FNeg f) = negate <$> encode' f
encode' f@FVar {} = getName f
getName :: Formula → Encoder Int
getName f = do
  state@EncoderState {formulaNames = names} ← get
  case f `Map.lookup` names of
    Just i → return i
    Nothing → do
      let name = Map.size names + 1
      put state {formulaNames = Map.insert f name names}
      return name

type Encoder = State EncoderState
data EncoderState = EncoderState
  { formulaNames :: Map.Map Formula Int
  , cnfRepr :: Set.Set [Int]
  }
```

```
prettyPrint :: EncoderState → IO ()
prettyPrint EncoderState {formulaNames = names, cnfRepr = cnf} = do
  originalCount ← sequence
    [case key of
       FVar name → putStrLn ("c " ++ name ++ " = " ++ show value) >> return 1
       _ → return 0
    | (key, value) ← Map.toList names
    ]
  let totalCount = Map.size names
  putStrLn $ "c $root = " ++ show totalCount
  putStrLn $ "p cnf " ++ show totalCount ++ ' ' : show (length cnf)
  sequence_
    [sequence_ [putStr (show item ++ " ") | item ← clause] >> putStrLn "0"
    | clause ← toList cnf
    ]
main :: IO ()
main = do
  contents ← T.getContents
  case parse formula "vstup" contents of
    Right f → do
      let result = execState (encode f) EncoderState {formulaNames = mempty, cnfRepr = mempty
      prettyPrint result
    Left e → print e
```