CS 558, Homework 3

Soumya Banerjee

November 8, 2010

1 Logic

```
module Main where

import qualified System.Environment
import IO


type Expr = [[Int]]

eval :: (Int -> Bool) -> Expr -> Bool
eval f = and . map (or . (map f))


{- function to evaluate a literal given a list of
 variable assignments. If no match found in list
of variable assignments, then it substitutes True
 for it -}
evalLiteral :: Int -> [(Int,Bool)] -> Bool
evalLiteral c [] = True
evalLiteral c ((v,boolval):l)
                | c == v  =  boolval
                | c == - v  =  not boolval
                | otherwise = evalLiteral c l


{- function to generate all the Boolean combinations-}
boolComb :: Int -> [[Bool]]
boolComb 0 = [[]]
boolComb (n + 1) = map (False:) bss ++ map (True:) bss
                        where bss = boolComb n

{- function to remove duplicates -}
removeDuplicates :: Eq a => [a] -> [a]
removeDuplicates [] = []
removeDuplicates (x:xs) = x:removeDuplicates (filter (/= x) xs)

{- function that given the expression generates
all the combinations -}
exhaustBool :: Expr -> [[(Int,Bool)]]
exhaustBool expr = map (zip vs) (boolComb (length vs))
                        where vs = map abs (removeDuplicates (concat expr))

convertTuple = map (\(u,v) -> if v then u else negate u)

{- function to return the satisfiability and
assignment -}
```

```
satisfiable' :: Expr -> [[(Int,Bool)]] -> (Bool,[Int])
satisfiable' _ [] = (False,[])
satisfiable' expr (l:ls)
        | eval (flip evalLiteral l) expr = (True,convertTuple l)
        | otherwise = satisfiable' expr ls

{- function to test satisfiability -}
satisfiable :: Expr -> Bool
satisfiable expr = fst (satisfiable' expr ls)
                       where ls = exhaustBool expr

readCNFFile :: String -> IO Expr
readCNFFile sourceFile = do y <- readFile sourceFile
                            return (parse y 0)


{- function to read a line and make a clause out of it -}
createClause [] = ([],[])
createClause (x:y:xs)
        | x == '\n' = ([],(y:xs))
        | x == ' ' = (fst (createClause (y:xs)), snd (createClause (y:xs)))
        | x == '0' = (fst (createClause (y:xs)), snd (createClause (y:xs)))
        | x == '-' = ((negate ((fromEnum y) - 48)):(fst (createClause xs)),
                                        snd (createClause xs))
        | otherwise = ( ((fromEnum x) - 48):(fst (createClause (y:xs))),
                                        snd (createClause (y:xs)))
createClause (x:xs)
        | x == '\n' = ([],xs)
        | x == ' ' = (fst (createClause xs), snd (createClause xs))
        | x == '0' = (fst (createClause xs), snd (createClause xs))
        | otherwise = ( ((fromEnum x) - 48):(fst (createClause xs)),
                                        snd (createClause xs))

{- function to take a string and make a formula
out of it -}
createFormula [] = []
createFormula (x:xs)
                | x == ' ' = createFormula xs
                | otherwise = [(fst p)] ++ (createFormula (snd p))
                                where p = createClause (x:xs)


parse' (x:xs)
                | x == '\n' = createFormula xs
                | otherwise = parse' xs

{- function to take a string and return the
```

```
expression/formula. commentLine is a flag
which stores the state of whether the curent
line is a comment line or not  -}
parse (x:xs) commentLine
                | (x == 'c' && commentLine == 0) = parse xs 1
                | (x == 'p' && commentLine == 0) = parse' xs
                | x == '\n' = parse xs 0
                | otherwise = parse xs commentLine

{- function to parse the assignment in clause
format and produce a string -}
parseOutput :: [Int] -> String
parseOutput [] = ['0']
parseOutput (x:xs)
        | x > 0 = ( (toEnum (x + 48) :: Char) : ([' '] ++ (parseOutput xs)) )
        | x < 0 = ( '-' : (toEnum ((abs x) + 48) :: Char) : ([' '] ++ (parseOutput xs)) )


main :: IO ()
main =
    do
        args <- System.Environment.getArgs
        let sourceFile = args !! 0
        let destfile = args !! 1
        expr <- readCNFFile sourceFile
        let y = satisfiable' expr (exhaustBool expr)
        if fst y then writeFile destfile ("SAT\n" ++ (parseOutput (removeDuplicates (snd y))))
        else writeFile destfile ("UNSAT\n" ++ (parseOutput (removeDuplicates (snd y))))
        eval(flip evalLiteral [(1,True),(2,False),(3,False),(4,False)]) [[-1, 2, 4], [-2, -3]]
        eval(flip evalLiteral [(1,True),(2,False),(3,False),(4,True)]) [[-1, 2, 4], [-2, -3]]
        satisfiable [[-1, 2, 4], [-2, -3]]
        satisfiable [[2], [-2]]
        readCNFFile "cnftest.txt"


{- OUTPUT

*Main> eval (flip evalLiteral [(1,True),(2,False),(3,False),(4,False)]) [[-1, 2, 4], [-2, -3]]
False
*Main> eval (flip evalLiteral [(1,True),(2,False),(3,False),(4,True)]) [[-1, 2, 4], [-2, -3]]
True


*Main> satisfiable [[-1, 2, 4], [-2, -3]]
True
*Main> satisfiable [[2], [-2]]
False
```

```
*Main> readCNFFile "cnftest.txt"
[[1,-3],[2,3,-1]]


Input file:- cnftest.txt
c  simple_v3_c2.cnf
c
p cnf 3 2
1 -3 0
2 3 -1 0

mack:~/cs557/hw3> ./hw32 "cnftest.txt" "cnfoutput.txt"
mack:~/cs557/hw3> cat cnfoutput.txt
SAT
-1 -3 -2 0

-}
```

2 Games

```
{- since I ran out of time, I just wrote
down the base case for the opening move
for Red. It should move into d1 -}

type Board = [[Maybe Player]]

data Player = PRed | PGreen
                deriving (Eq, Show)

heuristicStrategyForRed :: Board -> Board
{- opening move from empty board, move to middle position (d1)-}
heuristicStrategyForRed [[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing]] =
[[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Just PRed],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
```

```
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing]]

main :: IO ()
main =
    do
        print (heuristicStrategyForRed
([[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing]]))


{- Output

*Main> heuristicStrategyForRed
([[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing]])
[[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Just PRed],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing],
[Nothing,Nothing,Nothing,Nothing,Nothing,Nothing]]

-}
```