Homework 7

CS3490: Programming Languages

Due: Thursday October 20, 11PM

1. Reading

- Read Chapter 7 of the Haskell tutorial, at
  `http://learnyouahaskell.com/modules`

  Make a list of 5 functions from the List and Char library that you think will be most useful for writing programs that work with structured data.

- Read Chapter 8 of the Haskell tutorial, at
  `http://learnyouahaskell.com/making-our-own-types-and-typeclasses`

2. Programming

In this assignment, you will work with an extension of the datatype of Boolean formulas (Propositions) that includes new operations of implication, equivalence (if-and-only-if), and exclusive or.

```
type Vars = String
data Prop = Var Vars | Const Bool | And Prop Prop | Or Prop Prop | Not Prop
          | Imp Prop Prop | Iff Prop Prop | Xor Prop Prop
  deriving (Show,Eq)

prop1 = Var "X" `And` Var "Y"                    --    X /\ Y
prop2 = Var "X" `Imp` Var "Y"                    --    X -> Y
prop3 = Not (Var "X") `Or` (Var "Y")             --   !X \/ Y
prop4 = Not (Var "X") `Iff` Not (Var "Y")        --  !X <-> !Y
```

You are allowed to use solutions to the previous homework on propositional logic (the "take-home" midterm) in solving the following problems.

2.1. Evaluator

Extend the evaluator to account for the new operators.

```
type Env = [(Vars,Bool)]
eval :: Env -> Prop -> Bool
```

For negation, conjunction, and disjunction, you should use the standard truth tables to define the meaning of these operations.

The truth tables for the new operations of implication, equivalence, and eXclusive OR are given below:

X	Y	$X \to Y$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$

X	Y	$X \leftrightarrow Y$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\top$

X	Y	$X \oplus Y$
$\top$	$\top$	$\bot$
$\top$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\bot$

2.2. Lexical Analysis

Define the following datatypes for operators and tokens.

```
-- binary operators
data BOps = AndOp | OrOp | ImpOp | IffOp | XorOp
  deriving (Show,Eq)

-- the type of tokens
data Token = VSym Vars | CSym Bool | BOp BOps | NotOp | LPar | RPar
           | PB Prop  -- a token to store parsed boolean expressions
  deriving (Show,Eq)
```

Write a function `lexer :: String -> [Token]` which converts a string to a list of tokens. The representation of operators on the level of strings is defined in the following table.

Constructor	Representation
And	`/\`
Or	`\/`
Not	`!`
Imp	`->`
Iff	`<->`
Xor	`<+>`
Const True	`tt`
Const False	`ff`
Var X	Any string `X` starting with a capital letter and followed by zero or more alphanumeric characters, e.g. `X1`

Caution!! In Haskell, the backspace character `\` must be escaped at the level of strings. For example, to match a string beginning with `/\`, you should use the code

```
match ('/':'\\':s) = ....

GHCI> putStrLn "X \\/ Y"
X \/ Y
```

Internally, the string `"/\\"` is still a list containg two elements: the character `/`, represented `'/'`, and the character `\`, represented `'\\'`. The backslash should *not* be escaped by the user when they enter a test string via `getLine` function.

```
lexer "! (X \\/ Y)" = [NotOp, LPar, VSym "X", BOp OrOp, VSym "Y", RPar]
```

2.3. Parser

Write a parsing function `parseProp :: [Token] -> Prop` that takes as input a list of tokens and produces an element of the `Prop` datatype.

You can use a variation of the shift–reduce parser `sr` implemented in class.

```
parseProp [NotOp, LPar, VSym "X", BOp OrOp, VSym "Y", RPar]
  = Not (Or (Var "X") (Var "Y"))
```

2.4. Searching for a satisfying truth-assignment

Write a function `findSat :: Prop -> Maybe Env` which returns `Just s` if there is an environment `s` that causes the given formula to evaluate to `True`.

Otherwise, it should return `Nothing`

```
findSat prop1             = Just [("X",True),("Y",True)]
findSat (Not prop2)       = Just [("X",True),("Y",False)]
findSat (Not (Var "X"))   = Just [("X",False)]
findSat (Xor prop4 prop4) = Nothing
```

(*Hint.* You can approach this problem similarly to how you implemented the satisfiability checker in the previous assignment. This involves writing helper functions to get the variables, generating all possible assignments, and then searching through this list using a predicate that involves the `eval` function.)

2.5. Putting it all together

Write a function `solve :: String -> String` which takes a textual representation of a formula, analyses and parses it, and then either returns the string `"Satisfiable."` or returns the string `"No solution."`, depending on the result of the `findSat` function.

```
solve "X1 /\\ !Y2"     = "Satisfiable."
solve "(X -> Y) <-> X" = "Satisfiable."
solve "X <+> X"        = "No solution."
solve "X <+> ff"       = "Satisfiable."
```