Test 1

The take-home part

CS3490: Programming Languages

Due: Sunday, October 2

1. Generating strings

Define the following functions. Do not use any library functions.

1. `mapAppend :: (a -> [b]) -> [a] -> [b]` applies a function producing lists to every element of the list, and then appends all the lists together.

   ```
   mapAppend show [1,23,456] = "123456"
   mapAppend (\c -> c : " ") "hello" = "h e l l o "
   mapAppend (\s -> [ take i s | i <- [1..3]]) ["hello","there","world"]
     = ["h","he","hel","t","th","the","w","wo","wor"]
   ```

2. `addLetter :: Char -> [String] -> [String]` attaches the given letter in front of every string in the list.

   ```
   addLetter 'x' ["hey","again"] = ["xhey","xagain"]
   ```

3. `addLetters :: [Char] -> [String] -> [String]` runs the previous function with ever character in the given list, and puts the results together in one list.

   ```
   addLetters "abc" ["hey","again"]
       = ["ahey","aagain","bhey","bagain","chey","cagain"]
   ```

4. `makeWords :: [Char] -> Integer -> [String]` generates every possible word of a given length from a given set of characters.

   ```
   makeWords "abc" 2 = ["aa","ab","ac","ba","bb","bc","ca","cb","cc"]
   makeWords "ab" 4 = ["aaaa","aaab","aaba","aabb","abaa","abab","abba",
     "abbb","baaa","baab","baba","babb","bbaa","bbab","bbba","bbbb"]
   ```

2. Propositional logic

Define the following datatype of Boolean formulas (Propositions):

```
type Vars = String
data Prop = Var Vars | Const Bool | And Prop Prop | Or Prop Prop | Not Prop
  deriving Show
```

In the following examples, the single quotes are *backticks*:

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

In this problem, you will implement an algorithm which decides whether a given proposition is *satisfiable*. You will do this by constructing a truth table, which will require you to implement the following functions. You may use whichever library functions you find. You may also need to define your own auxiliary "helper" functions.

1. Write a function `fv :: Prop -> [Vars]` which lists all the variables occurring in a given formula. *Each variable should only be listed once*.

   ```
   fv prop1 = ["X","Y"]
   fv prop2 = ["X","Y"]
   fv (prop1 `And` prop2) = ["X","Y"]
   fv (Var "X" `Or` Var "Z") = ["X","Z"]
   ```

2. Write a function `lookUp :: Vars -> [(Vars,Bool)] -> Bool` which takes a key and a list of key–value pairs, and returns the value matching the given key

   ```
   lookUp "X" [("X",True),("Y",False)] = True
   lookUp "X" [("W",False),("X",True),("Y",False)] = True
   lookUp "Y" [("W",False),("X",True),("Y",False)] = False
   ```

3. Write the evaluator `eval :: [(Vars,Bool)] -> Prop -> Bool`.

   Given a list of variable–value pairs `env`, and a propositional formula `f : Prop`, the function `eval env f` returns `True` if the formula is true when all the variables are replaced by their values listed in `env`. In writing the evaluator, you should use the standard truth table semantics of `And`, `Or`, and `Not`.

   In implementing this function, you will need to have a way to look up the value associated to a variable in a list of pairs. If `env` does *not* provide a value for all the variables occurring in the formula `f`, the result of the function is undefined. Although, for debugging purposes, you may wish to check this condition explicitly. You could implement such a test via the expression `all (`elem` (map fst env)) (fv f)`.

```
eval [("X",False),("Y",True) ] prop1 = False
eval [("X",False),("Y",True) ] prop2 = True
eval [("X",False),("Y",True) ] prop3 = True
eval [("X",True) ,("Y",False)] prop3 = False
```

4. Write a function `evalAll` which, given a formula `f` and a *list of environments* `envs :: [[(Vars,Bool)]]`, returns `True` if evaluating `f` in *some element of* `envs` returns `True`. Otherwise, it should return `False`. This function should have type

   evalAll :: Prop -> [[(Vars,**Bool**)]] -> **Bool**

   EXAMPLE:

   ```
   evalAll prop1 [[("X",False),("Y",True)]]                        = False
   evalAll prop1 [[("X",False),("Y",True)],[("X",True),("Y",True)]] = True
   evalAll prop2 [[("X",False),("Y",True)]]                        = True
   evalAll prop2 []                                                = False
   evalAll prop4 [[("X",False),("Y",True)],[("X",True),("Y",True)]] = True
   evalAll prop4 [[("X",True) ,("Y",True)]]                        = False
   ```

5. The last piece you need is a function that, given a list of variables `vars :: [Vars]`, generates the list of *all possible environments* for these variables.

   Intuitively, you should think of this function as generating the "input rows" to the truth table, where every combination of True/False values is assigned to the variables given in the input list. This function has type

   genEnvs :: [Vars] -> [[(Vars,**Bool**)]]

   EXAMPLE:

   ```
   genEnvs ["X"]     = [[("X",True)],[("X",False)]]
   genEnvs ["X","Y"] = [[("X",True),("Y",True)],[("X",True),("Y",False)],[("X",False),("Y",True)],[("X",False),("Y",False)]]
   genEnvs []        = [[]]
   ```

   (*Hint.* First write a function which, given a *list* of environments, adds one more variable binding to *each* element of the list:

   ```
   extend ("Y",True) [[("X",True)],[("X",False)]] = [[("Y",True),("X",True)],[("Y",True),("X",False)]]
   ```

   Then use this function recursively twice for every new variable.)

6. With the functions above, it is now a straightforward matter to implement `sat :: Prop -> Bool` that checks whether the given formula is *satisfiable*: it returns True if there is some assignment to the variables that makes the formula evaluate to `True`.

   ```
   sat    (Var "X" `And` Var "Y")         = True
   sat    (Var "X" `And` (Not $ Var "X")) = False
   ```

Remark. The problem of deciding satisfiability is *NP-complete*. This means that if you find an algorithm that solves this problem and whose running time is polynomially proportional to the size of the input, then you can get $1,000,000. For more, see the wiki article: `https://en.wikipedia.org/wiki/Boolean_satisfiability_problem`