Homework 8:
Propositional Calculator Frontend

CS 3490: Programming Languages

Description

In this assignment, you will create a front-end for the propositional logic calculator you built in the previous assignment.

As always, you are allowed to adapt code from the lecture files to use in your solution file.

After all the import statements at the top, your file must contain the following line (case sensitive):

```
module Prop where
```

1. Setup

To begin with, you should use your solution to the previous homework, the propositional logic calculator. Alternatively, you can start with Dr. Polonsky's solution file posted on asulearn.

We will work with the same grammar of propositional formulas:

$$\mathbb{P} \quad ::= \quad \mathbb{V} \mid \mathbb{B} \mid \mathbb{P} \wedge \mathbb{P} \mid \mathbb{P} \vee \mathbb{P} \mid \neg\mathbb{P} \mid \mathbb{P} \to \mathbb{P} \mid \mathbb{P} \leftrightarrow \mathbb{P} \mid \mathbb{P} \oplus \mathbb{P}$$

which is represented in Haskell with the datatype

```
data Prop = Var Vars | Const Bool | And Prop Prop | Or Prop Prop
          | Not Prop | Imp Prop Prop | Iff Prop Prop | Xor Prop Prop
```

2. Read-eval-print loop

Write a `main` function for your program that will query the user to input commands to the calculator.

The primary difference between the calculator implemented in class and the front-end you have to implement here is the type of the environment used in the main loop.

Whereas the class calculator used the variable `env :: [(Vars,Value)]`, which assigned variables to integers, here you will use a parameter `def` of type `[(String,Prop)]` that will store *definitions* associating a *name* to a *proposition*.

It is convenient to introduce the following type synonyms:

```
type Name = String
type Def = [(Name,Prop)]
```

The `main` read-eval-print loop (REPL) will allow the user to associate a name to a proposition (boolean expression), and to execute commands associated to that name.

`let f = e`

> assigns the name `p` to the propositional formula `e`.
>
> This should result in the list of definitions being updated with the new pair `(f,e)`, where `f :: Name` is the string representing the name of the proposition, and `e :: Prop` is the parsed expression that will be associated to that name.
>
> Notice that, if the name `f` was already present in the look-up table, then it should be updated to point to the new element of `Prop`.

`vars f`

> returns all the propositional variables occurring in the formula `f`.
>
> *Repeated variables should only be listed once!*

`sat f`

returns whether the expression named by `f` is satisfiable or not.

That is, it tells where there exists a satisfying truth assignment to the variables of `f`.

The output should be a string `"Yes"` or `"No"`.

`tauto f`

returns whether the expression named by `f` is a tautology.

That is, it tells whether every assignment makes the formula evaluate to true.

Again, the output should be `"Yes"` or `"No"`.

`solve f`

search for a satisfying variable assignment.

If found, print it out. Otherwise, display the string `"No solution"`.

An assignment looks like a usual list of variable–value pairs, in this case `[(String,Bool)]`.

`eq f g`

states whether two formulas are *logically equivalent*.

Two formulas are equivalent if they have the same truth-conditions.

That is, they are equivalent if they are satisfiable in exactly the same truth assignments.

(*Hint.* You can write a helper function that generates all the environments for the variables occurring in either formula, and evaluates both formulas in each environment. Alternatively, you can check whether another formula involving `f` and `g` is a tautology.)

`subst x with f in g`

prints out the formula `g` but where every occurrence of the variable `x` has been replaced by the formula `f`.

If the variable `x` does not occur in `g` at all, then the effect should be the same as printing out the formula `g`. (It is unchanged by the substitution.)

`load filename`

load a list of definitions from a file.

These new definitions should reset those that were already present in the main loop from previous interaction with the user.

(Do the next part of the assignment before implementing this.)

`quit`

exit the program.

3. File I/O

The input file should have extension `.prop`

It will consist of a list of definition that look like as follows:

```
f = X /\ Y
g = f -> X
h = (g <-> g) \/ (Y -> f)
...
```

Every line begins with a `name`, followed by a formula.

There is no lexical difference between variables and names: they can all be analyzed as `VSym`.

Write a function

```
parseLines :: [String] -> Def
```

which, given the list of lines in a file, produces a table of definitions associating each name to a formula.