Using QuickCheck to develop a SAT solver

The Davis-Putnam-Logemann-Loveland algorithm is a method for deciding the satisfiability of propositional logic formulae. Although the SAT problem is NP-complete, it is still remarkably amenable to automation, and high-performance SAT-solvers are heavily used in modern software verification, constraint solving, and optimization. Your task in this problem is to implement one of the basic ideas behind the DPLL algorithm and check your implementation using QuickCheck.

We'll lead you through the main steps, but if you're not already familiar with the basics of SAT solving you may need to do a little background reading about the basic ideas in DPLL.

- DPLL Wikipedia page

Throughout, try to use library functions to make your code short and elegant. None of the requested function bodies should take much more than a dozen or so lines. Style counts!

```
> --------------------------------------------------------------------------
```

```
> module Sat where
```

In this problem, we will use a data structure from Haskell's standard library, implementing Finite Maps. The two import declarations below say: (1) import the type `Map` from the module `Data.Map` as an unqualified identifier, so that we can use just `Map a b` as a type; and (2) import everything else from `Data.Map` as qualified identifiers, written `Map.member`, etc.

This data structure is defined in the `containers` Haskell package (included with GHC). For a short introduction, see the containers tutorial documentation.

```
> import Data.Map (Map)
> import qualified Data.Map as Map
```

We will also make other library functions available. You can use any functions in these modules, as well as any other modules in the `base` libraries to complete the homework assignment.

```
> import qualified Data.List as List
> import qualified Data.Maybe as Maybe
```

Finally, we import definitions for HUnit and QuickCheck.

```
> import Test.HUnit (Test(..), (~:), (~?=), runTestTT, assertBool)
> import Test.QuickCheck
```

```
> --------------------------------------------------------------------------
> -- Basic types
```

The DPLL algorithm works on formulae that are in Conjunctive Normal Form (CNF), i.e. formulae that consist of a conjunction of *clauses*, where each clause is a disjunction of *literals*, i.e. positive or negative propositional variables. For example,

```
(A \/ B \/ C) /\ (not A) /\ (not B \/ C)
```

is a CNF formula. This formula is satisfiable: if we set the variable `A` to be `False` and `C` to be `True`, we can rewrite the formula to be

```
(False \/ B \/ True) /\ (not False) /\ (not B \/ True)
```

Each clause of the resulting formula has a true literal, so we can see this assignment satisfies the formula (and that it doesn't matter whether `B` is `True` or `False`).

We represent CNF formulae with the following type definitions.

```
> -- | An expression in CNF (conjunctive normal form) is a conjunction
> -- of clauses. We store these clauses in the conjunction in a list.
> newtype CNF = Conj { clauses :: [Clause] } deriving (Eq, Ord, Show)
```

```
> -- | A clause is a disjunction of a number of literals, again storing
> -- each literal in a list.
> newtype Clause = Disj { lits ::  [Lit] } deriving (Eq, Ord, Show)
```

```
> -- | A literal is either a positive or a negative variable
> data Lit = Lit { polarity :: Bool, var :: Var } deriving (Eq, Ord, Show)
```

```
> -- | A variable is just a character
> newtype Var = Var Char
>   deriving (Eq, Ord, Show)
```

```
> -- A few variables for test cases
> vA, vB, vC, vD :: Var
> vA = Var 'A'
> vB = Var 'B'
> vC = Var 'C'
> vD = Var 'D'
```

Here's how the formula from above is represented:

```
> exampleFormula :: CNF
> exampleFormula = Conj [
>     Disj [Lit True vA, Lit True vB, Lit True vC],
>     Disj [Lit False vA],
>     Disj [Lit False vB, Lit True vC]]
```

More generally, because clauses are *disjunctions* of literals, a clause that is merely the empty list represents the truth value "False".

```
> falseClause :: Clause
> falseClause = Disj []
```

This makes sense because `False` is an identity element for disjunction.

On the other hand, CNF formulas are *conjunctions* of clauses, a formula that is merely the empty list represents the truth value "True".

```
> trueCNF :: CNF
> trueCNF = Conj []
```

Again, this follows because `True` is the identity element for conjunction.

```
> --------------------------------------------------------------------------
```

The next few operations allow us to work with formulae, clauses, literals and variables.

First observe that `clauses` will extract the list of clauses from a `CNF`, `lits` will extract the list of literals from a `Clause`, and `var` will extract the variable from a `Literal`.

We also have two functions for working with the polarity of a literal.

```
> -- | Is the literal positive?
> isPos :: Lit -> Bool
> isPos = polarity
```

```
> -- | Negate a literal
> neg :: Lit -> Lit
> neg (Lit b x) = Lit (not b) x
```

Variables are enumerable. However, only the first 26 will print nicely.

```
> instance Enum Var where
>   toEnum i         = Var (toEnum (i + fromEnum 'A'))
>   fromEnum (Var v) = fromEnum v - fromEnum 'A'
```

```
> -- | A long list of variables
> allVars :: [ Var ]
> allVars = [ vA .. ]
```

```
> --------------------------------------------------------------------------
```

Sometimes we need to know about all of the variables that appear in a particular formula. We can use a finite map structure to calculate this information. (You'll need to refer to the documentation for the `Data.Map` module to complete this part.)

```
> -- | The number of times each variable appears in the formula
> -- >>> countVars exampleFormula
> -- fromList [(Var 'A',2),(Var 'B',2),(Var 'C',2)]
> countVars :: CNF -> Map Var Int
> countVars = undefined
```

```
> -- | All of the variables that appear anywhere in the formula, in sorted order
> -- >>> vars exampleFormula
> -- [Var 'A',Var 'B',Var 'C']
> vars :: CNF -> [Var]
> vars = undefined
```

Here are the unit tests that correspond to the doctests above. Make sure that these tests pass before continuing to the next step.

```
> testCountVars :: Test
> testCountVars = "countVars" ~:
>   countVars exampleFormula ~?= Map.fromList [(vA, 2), (vB, 2), (vC, 2)]
```

```
> testVars :: Test
> testVars = "vars" ~:
>    vars exampleFormula ~?= [vA, vB, vC]
```

```
> --------------------------------------------------------------------------
```

Of course, most of the testing we will do in this homework will use QuickCheck.

To do that, we need to be able to generate arbitrary formulae. The following generators should be parameterized by the number of distinct variables to use. When you are testing solvers below, you'll find that changing this number affects the efficiency of certain solvers and also the distribution of satisfiable random formulae.

For example, if we sample from `genVar` below, we should only see three different variables.

```
*Sat> xs <- sample' (genVar 3)
*Sat> xs
[Var 'A',Var 'A',Var 'C',Var 'C',Var 'A',Var 'C',Var 'B',Var 'A',Var 'C',Var 'A',Var 'A']
```

You should use the functions in the [QuickCheck library] (https://hackage.haskell.org/package/QuickCheck-2.14.2/docs/Test-QuickCheck-Gen.html) to define these generators.

```
> -- | Generate a random variable (limited to the first `n` variables).
> genVar      :: Int -> Gen Var
> genVar    n | n < 1 = error "Must supply a positive number to genVar"
> genVar n = undefined
```

```
> -- | Generate a random literal chosen from at most `n` distinct variables.
> genLit      :: Int -> Gen Lit
> genLit n = undefined
```

```
> -- | Generate a random Clause with at most `n` distinct variables.
> -- The clause may be of any length
> genClause   :: Int -> Gen Clause
> genClause n = undefined
```

```
> -- | Generate a random CNF with at most `n` distinct variables.
> -- The formula may be of any length
> genCNF      :: Int -> Gen CNF
> genCNF n = undefined
```

Of course, we should test these generators. Here are two unit tests that we can use to make sure that we don't generate formulae with too many variables.

```
> -- make sure that genVars produces the right number of variables.
> testGenVars :: Test
> testGenVars = "genVars" ~: do
>    xs <- sample' (genVar 3)
>    return $ length (List.nub xs) == 3
```

```
> -- make sure that arbitrary formulae don't contain too many variables.
> testGenCNF :: Test
> testGenCNF = "genCNF num vars" ~: do
>   xs <- sample' (genCNF defaultNumVariables)
>   return $ all (\c -> length (countVars c) <= defaultNumVariables) xs
```

And here is one that makes sure that we are not always generating the same formulae. Because of the randomness involved, this test could fail even with a correct implementation. But that is unlikely.

```
> -- make sure that we generate different formulae.
> testGenCNFdiff :: Test
> testGenCNFdiff = "genCNF diff" ~: do
>   xs <- sample' (genCNF defaultNumVariables)
>   return (length (List.nub xs) >= 8)
```

We use these generators in our `Arbitrary` instances along with appropriate definitions for `shrink`.

```
> defaultNumVariables :: Int
> defaultNumVariables = 7
```

```
> instance Arbitrary Var where
>   arbitrary = genVar defaultNumVariables
>   shrink v | v == vA   = []
>            | otherwise = [ vA .. pred v ]
```

```
> instance Arbitrary Lit where
>   arbitrary = genLit defaultNumVariables
>   shrink (Lit b v) = map (`Lit` v) (shrink b) ++
>                      map (Lit b) (shrink v)
```

```
> instance Arbitrary Clause where
>    arbitrary = genClause defaultNumVariables
>    shrink (Disj l) = map Disj (shrink l)
```

```
> instance Arbitrary CNF where
>    arbitrary = genCNF defaultNumVariables
>    shrink (Conj x) = map Conj (shrink x)
```

```
> --------------------------------------------------------------------
> -- Satisfiable and unsatisfiable formulae
```


Our example formula is said to be *satisfiable* because it is possible to find an assignment of truth values to variables -- namely


```
A |-> False
B |-> True
C |-> True
```


— that makes the example formula true. On the other hand, this formula

```
> --  A /\ (not A)
> unSatFormula :: CNF
> unSatFormula = Conj [Disj [Lit True vA], Disj [Lit False vA]]
```

is *unsatisfiable* because there is no such assignment.

An assignment of truth values to variables is called a *valuation*. (In logic, valuations usually assign a truth value to *all* variables of a formula. Here we will do things a little bit differently and define a valuation to be a map from *some* variables to truth values.)

```
> -- | Assignments of values to (some) variables
> type Valuation = Map Var Bool
```

To make our code a more readable, we rename some of the operations from the `Data.Map` library to be specific to `Valuations`.

```
> -- | No bindings
> emptyValuation :: Valuation
> emptyValuation = Map.empty
```

```
> -- | Add a new binding
> extend :: Var -> Bool -> Valuation -> Valuation
> extend = Map.insert
```

```
> -- | Check the value of a variable
> value :: Var -> Valuation -> Maybe Bool
> value = Map.lookup
```

```
> -- | Create a valuation from a given list of bindings
> fromList :: [(Var,Bool)] -> Valuation
> fromList = Map.fromList
```

For instance, the valuation above is represented thus:

```
> exampleValuation :: Valuation
> exampleValuation = Map.fromList [(vA, False), (vB, True), (vC, True)]
```

We say that a valuation *satisifes* a CNF formula if the valuation makes the formula true.

```
> satisfiesLit :: Valuation -> Lit -> Bool
> satisfiesLit a lit = a Map.!? var lit == Just (polarity lit)
```

```
> satisfies :: Valuation -> CNF -> Bool
> satisfies a cnf = all (any (satisfiesLit a) . lits) (clauses cnf)
```

Take a moment to look at the definition of `satisfies` and consider the following two formulae.

- This first formula is a conjunction of zero clauses, all of which must be satisfied for the formula to be true. So this formula will be satisfied by any valuation, including the empty one. We can view this formula as equivalent to the logical formula "True".

```
> validFormula :: CNF
> validFormula = Conj []
```

- On the other hand, `anotherUnsatFormula` below is the conjunction of a single clause. That clause must be satisfied in order for the whole formula to be true. However, that clause is a disjunction; there must be some true literal in the clause for it to be satisfied, and there isn't. So this formula cannot be satisfied by any valuation. We can view this formula as equivalent to the logical formula "False".

```
> anotherUnsatFormula :: CNF
> anotherUnsatFormula = Conj [ Disj [] ]
```

Let's create a few tests for `satisfies`. Our example formula is satisfied by the example valuation. (Add a few more tests of formulae and their satisfying valuations to the list below.)

```
> testSatisfies :: Test
> testSatisfies = "satisfies" ~:  TestList
>      [ "exampleFormula" ~:
>          assertBool "" (exampleValuation `satisfies` exampleFormula)
>      , "another example" ~: assertBool "" (error "ADD your own test case here") ]
```

```
> --------------------------------------------------------------------------
```

Note that our unsatisfiable formula is not satisfied by ANY valuation. This is a property that we can check.

First note that only the variables that occur in a formula determine whether it is satisfiable. This seems obvious, but what it means is that that when looking whether formulae are unsatisfiable, we can restrict our attention only to possible valuations for the vars of a formula.

So, the first step is to implement the `makeValuations` function that constructs *all* valuations that assign values to a given list of variables.

```
> makeValuations :: [Var] -> [Valuation]
> makeValuations = undefined
```

To test your implementation, QuickCheck the following property stating that `makeValuations` is correct, in the sense that it has the right length (2^*n*, where *n* is the number of variables in the set) and all its elements are distinct.

```
> prop_makeValuations :: CNF -> Property
> prop_makeValuations p = length valuations === 2 ^ length ss
>                      .&&. allElementsDistinct valuations
>   where
>     valuations = makeValuations ss
>     ss = vars p
```

```
> allElementsDistinct :: Eq a => [a] -> Bool
> allElementsDistinct []     = True
> allElementsDistinct (x:xs) = x `notElem` xs
>                           && allElementsDistinct xs
```

With this definition, we can write a function that determines whether a formula is unsatisfiable.

```
> -- | A formula is unsatisfiable when there is no satisfying valuation
> -- out of all of the possible assignments of variables to truth values
> -- >>> unsatisfiable unSatFormula
> -- True
> unsatisfiable :: CNF -> Bool
> unsatisfiable p = not . any ( `satisfies` p) $ makeValuations (vars p)
```

```
> testUnsatisfiable :: Test
> testUnsatisfiable = "unsatisfiable" ~:  TestList
>      [ "unSatFormula"   ~: assertBool "" (unsatisfiable unSatFormula),
>        "exampleFormula" ~: assertBool "" (not (unsatisfiable exampleFormula))]
```

```
> --------------------------------------------------------------------------
> -- Simple SAT Solver
```

A *solver* is a function that finds a satisfying valuations for a given formula (assuming one exists).

```
> type Solver = CNF -> Maybe Valuation
```

The first solver you should implement is a simple combinatorial solver that basically tries all possible valuations and stops whenever it finds a satisfying valuation. In other words, it should search the list of all valuations return the first one that satisfies the given formula. This is an expensive solver, especially if the formula contains many variables.

```
> sat0 :: Solver
> sat0 = undefined
```

To check that it works, QuickCheck the property `prop_satResultSound sat0`, stating that a successful result returned by a `sat0` is always a satisfying valuation. We'll also collect the percentage of formulae in each category, to give us more information about our formulae generationq.

```
> prop_satResultSound :: Solver -> CNF -> Property
> prop_satResultSound solver p = case solver p of
>         Just a  -> collect "sat" $ a `satisfies` p
>         Nothing -> collect "unsat" $ property True
```

A solver is also *complete* if whenever it fails to find a satisfying assignment, then that formula is unsatisfiable. We say that a solver is correct if it is both sound and complete.

```
> prop_satResultCorrect :: Solver -> CNF -> Property
> prop_satResultCorrect solver p = property $ case solver p of
>     Just a  -> a `satisfies` p
>     Nothing -> unsatisfiable p
```

This correctness property will always be expensive to test, so we separate it from soundness.

```
> --------------------------------------------------------------------------
> -- Instantiation
```

The simple solver we have just developed is very inefficient. One reason for this is that it evaluates the whole formula every time it tries a different valuation. Indeed, we can do much better: once we choose a truth value for a propositional variable, we can simplify the formula to take this choice into account.

For instance, imagine we have the CNF formula

```
(A \/ not B) /\ (not A \/ B \/ C).
```

If we instantiate `A` to `True`, then the formula becomes `(True \/ not B) /\ (False \/ B \/ C)`, which can be simplified to `(B \/ C)`.

Note: if a variable is not present in a formula, `instantiate` should return the formula unchanged.

Please implement the `instantiate` function:

```
> instantiate :: CNF -> Var -> Bool -> CNF
> instantiate = undefined
```

Informally, the correctness property for `instantiate` is that, if `s` is a formula and `v` a variable, `s` should be satisfiable iff either `instantiate s v True` or `instantiate s v False` is satisfiable. Use the simple satisfiability checker `sat0` to state this formally as a QuickCheck property. Use QuickCheck (in GHCi) to test whether your implementation of `instantiate` has this property.

```
> prop_instantiate :: CNF -> Var -> Bool
> prop_instantiate = undefined
```

Now use `instantiate` to write a sat solver that, at each step, chooses a variable and recursively tries instantiating it with both `True` and `False`.

The algorithm should proceed like this:

1. First, check if the formula is either obviously `satisfied` (returning an empty valuation if so) or `falsified` (returning `Nothing`). A formula has been satisfied when it is satisfied by the empty valuation (i.e. we have already instantiated enough variables in the formula that the others don't matter.) Alternatively, a formula is `falsified` when, like `anotherUnsatFormula`, it contains a clause with an empty disjunction.

2. Otherwise, choose one of the variables in the formula, instantiate it with both `True` and `False`, and see if either of the resulting formulae are satisfiable. If so, add an appropriate binding for the variable we instantiated to the resulting `Valuation` and return it.

IMPLEMENTATION NOTES:

- Make sure that your definition of `falsified` doesn't rely on the formula being *completely* instantiated. It is possible to cut off large parts of the search space by identifying unsatifiable formula as soon as possible.

- If your implementation determines that instantiating a variable with, say `True` produces a satisfying valuation, make sure that it does not also test whether instantiating that variable with `False` also works. You only need to find one valuation and can cut off the search at that point.

```
>
> sat1 :: Solver
> sat1 = sat where
>   sat = undefined
```

To check that it works, QuickCheck the property `prop_satResultSound sat1`, `prop_satResult sat1`, plus this one that says that it should agree with our previous solver (and any others that you can think of).

```
> prop_sat1 :: CNF -> Property
> prop_sat1 s = property (Maybe.isJust (sat1 s) == Maybe.isJust (sat0 s))
```

If you run this file, you'll see that `sat1` is significantly faster than `sat0` as the number of variables in the formula grows.

For efficient testing, we have hardwired `defaultNumVariables` so that none of the tests go too slowly. However, you can use the QuickCheck library function `forAll` to generate formulae that use more variables. For example, when testing with nine possible variables we found our implementation of `sat1` to be consistently more efficient than `sat0`

```
*Sat> :set +s
*Sat> quickCheckN 500 $ forAll (genCNF 9) (prop_satResultSound sat0)
+++ OK, passed 500 tests:
62.4% "sat"
37.6% "unsat"
(2.11 secs, 1,817,340,656 bytes)
*Sat> quickCheckN 500 $ forAll (genCNF 9) (prop_satResultSound sat1)
+++ OK, passed 500 tests:
62.6% "sat"
37.4% "unsat"
(0.46 secs, 475,496,272 bytes)
```

This is only the beginning. Modern SAT solvers employ a number of techniques to speed up this process including unit propagation, pure literal elimination, clause learning, and much more.

```
> --------------------------------------------------------------------------
> -- All the tests in one convenient place:
```

```
> quickCheckN :: Testable prop => Int -> prop -> IO ()
> quickCheckN n = quickCheckWith $ stdArgs { maxSuccess = n }
```

```
> main :: IO ()
> main = do
>     putStrLn "Unit tests:"
>     _ <- runTestTT $ TestList [testCountVars, testVars, testGenVars,
>                               testGenCNF, testGenCNFdiff, testSatisfies, testUnsatisfiable]
>     putStrLn "Quickcheck properties:"
>     putStrLn "prop_satResultSound sat0"
>     quickCheckN 500 $ prop_satResultSound sat0
>     putStrLn "prop_satResultCorrect sat0"
>     quickCheckN 500 $ prop_satResultCorrect sat0
>     putStrLn "prop_instantiate"
>     quickCheckN 500 prop_instantiate
>     putStrLn "prop_sat1"
>     quickCheckN 500 prop_sat1
>     putStrLn "prop_satResultSound sat1"
>     quickCheckN 500 $ prop_satResultSound sat1
>     putStrLn "prop_satResultCorrect sat1"
>     quickCheckN 500 $ prop_satResultCorrect sat1
```