2019 Haskell January Test
SAT Solving

This test comprises four parts. Any marks accumulated in Part IV are bonus marks and these will be added to the total obtained from Parts I-III, the maximum mark for which is 25. The total mark will be capped at a maximum of 25. The **2019 Haskell Programming Prize** will be awarded for the best overall solution.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of Haskell's various language features and predefined functions.

WARNING: The examples and test cases here are not guaranteed to exercise all aspects of your code. You are therefore advised to define your own tests to complement the ones provided.

1 Introduction

The *Satisfiability*, or SAT, problem is that of determining whether there are assignments to the variables of a propositional logic formula which make the formula $True$. SAT problems are key to many real-world applications that can be formulated in propositional logic, such as industrial product design, hardware/software verification and task scheduling.

In theoretical computer science the SAT problem is also important as it is an example of an *NP-complete* problem – a class of decision problems for which no efficient solution algorithm is known in the general case. Fortunately, there are some simple *heuristics*, i.e. rules that work well in practice without guaranteeing optimality, that can massively reduce the SAT solving time for many practical examples. This exercise concerns the famous DP (Davis-Putnam) algorithm which augments an exhaustive "brute force" search strategy with an additional heuristic called *unit clause propagation*[1] designed to reduce the size of the search space. Astonishingly, this very simple heuristic often leads to extremely short solution times when the brute-force strategy alone would render the problem essentially intractable.

Note that an alternative way to solve a SAT problem is to build a Binary Decision Diagram (BDD) from the given propositional formula and then determine whether any of the terminal leaves of the BDD is labelled $True$. This was the subject of a previous test.

2 Propositional logic

In this exercise you will be working with a subset of propositional logic formulae comprising just propositional variables, e.g. a, b, c, ..., each of which can assume the values $False$ or $True$, and the familiar logical connectives *not* ($\neg$), *and* ($\wedge$) and *or* ($\vee$). The implication ($\rightarrow$) and iff ($\Leftrightarrow$) connectives are omitted as they can be straightforwardly rewritten in terms of the other three. Some examples of well-formed propositional formulae are shown in the left column of Table 1. The operations $\neg$, $\wedge$ and $\vee$ have the usual meaning:

a	$\neg a$
$False$	$True$
$True$	$False$

a	b	$a \wedge b$	$a \vee b$
$False$	$False$	$False$	$False$
$False$	$True$	$False$	$True$
$True$	$False$	$False$	$True$
$True$	$True$	$True$	$True$

2.1 Conjunctive normal form

The DP algorithm assumes that propositional formulae are in *Conjunctive Normal Form*, or CNF, i.e. in the form of a conjunction ($\wedge$) of zero or more *clauses*, each of which is a disjunction ($\vee$) of zero or more *literals*, each of which is either a variable or its negation, e.g. a or $\neg a$. For example, the middle column of Table 1 shows the CNF equivalents of the example formulae in the left column. Note that we can optionally drop the brackets when writing out nested conjunctions and disjunctions, although the examples in Table 1 retain them. For example, $A \wedge (B \wedge C)$ can be written $A \wedge B \wedge C$ and $(A \vee B) \vee C$ can be written $A \vee B \vee C$.

[1] The published algorithm also uses a second heuristic called *pure literal elimination*, but this is only beneficial in the boolean (yes/no) satisfiability problem. Here, we are interested in computing *all* satisfying assignments, so it is therefore omitted for simplicity.

Formula	CNF	Flattened Representation
$\neg(\neg a)$	a	`[[1]]`
$a \wedge b$	$a \wedge b$	`[[1],[2]]`
$\neg(\neg a \wedge \neg b)$	$a \vee b$	`[[1,2]]`
$\neg(b \vee (a \wedge \neg c))$	$\neg b \wedge (\neg a \vee c))$	`[[-2],[-1,3]]`
$(a \wedge (\neg(b \vee c))) \vee \neg d$	$(a \vee \neg d) \wedge ((\neg b \vee \neg d) \wedge (\neg c \vee \neg d))$	`[[1,-4],[-2,-4],[-3,-4]]`

Table 1: Some examples of propositional formulae (respectively `f1`, `f3`, `f4`, `f6` and `f8` in the template) and their CNFs

2.1.1 Conversion to NNF

In order to convert a propositional formula into CNF it is first necessary to convert it to *Negation Normal Form*, or NNF. To do this, you use De Morgan's laws and the law of double negation, so that all negations end up being attached only to variables:

$$\begin{aligned} \neg(A \vee B) &\equiv \neg A \wedge \neg B \\ \neg(A \wedge B) &\equiv \neg A \vee \neg B \\ \neg\neg A &\equiv A \end{aligned}$$

where A and B are arbitrary sub-formulae, which we shall sometimes refer to as *terms*. Note that these rules have to be applied recursively, as A and B may themselves contain sub-formulae (terms) to which the same rules must be applied.

2.1.2 Conversion to CNF

Once a formula is in NNF it is straightforward to convert it to CNF by distributing *Or*s inwards over *And*s via:

$$\begin{aligned} A \vee (B \wedge C) &\equiv (A \vee B) \wedge (A \vee C) \\ (A \wedge B) \vee C &\equiv (A \vee C) \wedge (B \vee C) \end{aligned}$$

2.2 Flattened CNF representation

When implementing the DP algorithm it is convenient to work with a *flattened* representation of CNFs, which we'll call `CNFRep`, which exploits the fact that CNFs are simply "*And*s of *Or*s". In this representation propositional variables will be encoded as positive integers, e.g. 3 for c, and their negation by their negative equivalent, e.g. -3 for $\neg c$. Each clause (disjunction) will be represented by a list of literals, each encoded as an integer as above, and the top-level conjunction will be represented by a list of these flattened clauses. For example, assuming that a, b, c and d are encoded as 1, 2, 3 and 4 respectively, the right column of Table 1 shows the flattened representations of the CNFs in the middle column. We adopt Haskell's list notation for obvious reasons!

3 Haskell types

For the purposes of this exercise you're going to work with the following representation for propositional formulae and CNFs, which is self-explanatory:

```
type Id = String

data Formula = Var Id
             | Not Formula
             | And Formula Formula
             | Or  Formula Formula
             deriving (Eq, Show)

type NNF = Formula

type CNF = Formula

type CNFRep = [[Int]]

type IdMap = [(Id, Int)]
```

These types are provided in a file called `Types.hs` which is separate from the main template.

Notice that, for simplicity, both `NNF` and `CNF` are synonymous with `Formula`. An assumption throughout is therefore that all NNFs and CNFs are well-formed in that they respect the definition of the respective normal form. Thus, for example, you may assume that an object with type `NNF` *will* be in Negation Normal Form even though formulae that are not in that form could be assigned the type `NNF`.

To help with testing, a function called `printF` is provided in the template, which will print a shorthand version of a given `Formula`, using the symbols `!`, `&` and `|` to represent $\neg$, $\wedge$ and $\vee$ respectively. Thus, for example:

```
*SAT> f1
Not (Not (Var "a"))
*SAT> printF f1
!!a
*SAT> f4
Not (And (Not (Var "a")) (Not (Var "b")))
*SAT> printF f4
!(!a & !b)
*SAT> f8
Or (And (Var "a") (Not (Or (Var "b") (Var "c")))) (Not (Var "d"))
*SAT> printF f8
((a & !(b | c)) | !d)
```

At this point you are in a position to complete Parts I and II. You may wish to work on these these before reading on.

4 The DP algorithm

Given a propositional formula in CNF the DP algorithm will determine whether there exists an assignment of variables to truth values, i.e. to $False$ or $True$, that will make the formula $True$. However, simply knowing that a formula is satisfiable isn't that useful without knowing which assignments, if any, satisfy it. This exercise therefore seeks to build a version of DP that computes *all* satisfying assignments of a given formula, should any such assignments exist.

An exhaustive "brute force" approach would be to try all possible variable assignments and check each in turn to see whether the formula is $True$ under the given assignment. Unfortunately, if there are n variables then there are 2^n possible assignments, so the problem is computationally intractable except for small values of n. As an example, the so-called n-queens problem is to place n queens on a chessboard in such a way that no two queens can take each other. On an 8×8 chess board there are 64 possible locations, so a propositional formula that describes all the constraints for a solution to be valid will involve 64 variables. A brute force approach would thus require all $2^{64} = 18446744073709551616$ possible assignments to be explored. To put this in perspective, if a billion assignments could be tested each second then it would take approximately 585 years(!) to compute all solutions, of which there are known to be just 92.

The DP algorithm is essentially the brute force approach but with an additional heuristic[2], *unit clause propagation*, designed to prune out parts of the search space so that, hopefully, only a tiny fraction of all possible assignments end up being considered. For example, the DP algorithm can compute all 92 solutions to the n-queens problem in under a second on most lab machines.

4.1 Unit clause propagation

Consider a formula in CNF comprising a conjunction of clauses, each representing a disjunction of literals. Clauses comprising just a single literal are called *unit clauses*. For example, in the formula $a \wedge ((b \vee c) \wedge \neg c)$, equivalently $a \wedge (b \vee c) \wedge \neg c$, both a and $\neg c$ are unit clauses, but $(b \vee c)$ is not.

The reasoning behind unit clause propagation goes as follows. Suppose we have a CNF formula, F, that contains a unit clause U, i.e a single literal comprising a propositional variable or its negation. In order for F to be satisfied, U must be $True$, otherwise F, which is a conjunction, would contain a $False$ term and would hence itself be $False$. Thus, the following deletion rules apply:

1. **Clause deletion**: If any clause (disjunction) in F contains U then that clause can be deleted, because: i. a disjunction of terms, at least one of which is $U = True$, must itself be $True$, i.e. $True \vee A \equiv A \vee True \equiv True$; ii. a conjunction of terms, at least one of which is $True$, is the same as the conjunction with the $True$ term(s) removed, because $True \wedge A \equiv A \wedge True \equiv A$. Notice that this rule also has the effect of deleting the unit clause itself (U) from F.

2. **Literal deletion**: If U is $True$ then $\neg U$ is $False$, e.g. if the unit clause $\neg c$ is $True$ then c must be $False$. Thus, if any of the other clauses in F contains the literal $\neg U$, then that literal can be deleted from the clause, because $False \vee A \equiv A \vee False \equiv A$.

4.2 Worked example

To illustrate the DP algorithm acting on a flattened CNF, consider `cnf2` defined in the template, which corresponds to the following propositional formula in CNF:

$$(\neg a \vee \neg d) \wedge ((b \vee (a \vee e)) \wedge ((c \vee \neg b) \wedge (d \wedge ((c \vee \neg e) \wedge ((\neg c \vee \neg g) \wedge ((a \vee (g \vee f)) \wedge (\neg c \vee f)))))))$$

or, equivalently:

$$(\neg a \vee \neg d) \wedge (b \vee a \vee e) \wedge (c \vee \neg b) \wedge d \wedge (c \vee \neg e) \wedge (\neg c \vee \neg g)) \wedge (a \vee g \vee f) \wedge (\neg c \vee f)$$

The flattened equivalent, `cnf2Rep`, is also defined in the template, thus:

[2] Actually two additional heuristics as originally described – see the footnote above.

```
*SAT> :t cnf2
cnf2 :: CNF
*SAT> cnf2
And (Or (Not (Var "a")) (Not (Var "d"))) (And (Or (Var "b") (Or (Var "a") (Var
"e"))) (And (Or (Var "c") (Not (Var "b"))) (And (Var "d") (And (Or (Var "c")
(Not (Var "e"))) (And (Or (Not (Var "c")) (Not (Var "g"))) (And (Or (Var "a")
(Or (Var "g") (Var "f"))) (Or (Not (Var "c")) (Var "f"))))))))
*SAT> printF cnf2
((!a | !d) & ((b | (a | e)) & ((c | !b) & (d & ((c | !e) & ((!c | !g) &
((a | (g | f)) & (!c | f))))))))
*SAT> cnf2Rep
[[-1,-4],[2,1,5],[3,-2],[4],[3,-5],[-3,-7],[1,7,6],[-3,6]]
```

Notice that the nested invocations of `And` and `Or` in the original `CNF` have been flattened into a single conjunction of eight clauses in the corresponding `CNFRep`. Each clause (a disjunction) has been flattened similarly, e.g. `Or (Var "c") (Not (Var "e"))` becomes `[3,-5]`. From here on we will work with the flattened representations (`CNFRep`s) only.

Unit clauses correspond to the literals associated with singleton lists in the `CNFRep` representation. In the example above, there is one such list, `[4]`, with associated unit clause `4`, corresponding to variable d in the original CNF. After propagating this using the deletion rules from Section 4.1, we end up with:

```
[[-1],[2,1,5],[3,-2],[3,-5],[-3,-7],[1,7,6],[-3,6]]
```

Notice that the clause `[4]` has been deleted by Rule 1 above because, by definition, it contains the literal `4`. In this case it is the only such clause and so is the only one to be deleted. The literal `-4` has also been removed from every clause by Rule 2 above, so the element `[-1,-4]`, has been replaced by `[-1]`. This is now a singleton list, so `-1` represents a new unit clause, corresponding to variable a in the original formula. The process therefore repeats, yielding:

```
[[2,5],[3,-2],[3,-5],[-3,-7],[7,6],[-3,6]]
```

This time the literal `1` has been removed from every clause by Rule 1 above. There were no occurrences of `-1` other than the unit clause itself, so Rule 2 did not apply.

At this point the unit clauses propagated have been `4` and `-1` and this has yielded a non-empty CNF in which there are no further unit clauses. When this happens the algorithm moves into "brute force" mode by picking one of the remaining variables, (any one will do) and then recursively invoking the DP algorithm *twice* – once assuming the variable to be $False$ and then again assuming it to be $True$. For the purposes of this exercise, the next variable to pick will be the one associated with the first literal in the current CNF. In this case it is `2`, corresponding to b in the original formula.

The order in which we consider the recursive calls is not important, as we are looking for *all* satisfying assignments, so let's first consider the case where we set b to $True$. This is done by adding the unit clause `2` to the above `CNFRep` in the form of a singleton list, for example at the front, as in `[[2],[2,5],[3,-2],[3,-5],[-3,-7],[7,6],[-3,6]]`, before recursing. Notice that this is equivalent to a propositional formula of the form $(b \wedge ...)$ which can only be $True$ when b is $True$, as required. Notice also that `2` is a (new) unit clause, so the recursive invocation of the DP algorithm will immediately remove it as part of the unit propagation process.

The recursive call thus proceeds as follows: first the unit clause `2` is propagated, yielding:

```
[[3],[3,-5],[-3,-7],[7,6],[-3,6]]
```

which exposes `3` as a new unit clause. Propagating this yields:

```
[[-7],[7,6],[6]]
```

There are now two unit clauses to choose from and it matters not which one you pick. If you propagate `-7` first then the result is:

```
[[6],[6]]
```

and propagating the unit clause `6` from either of these will knock out the other (Rule 1), yielding `[]`. An empty conjunction, i.e. `[]`, corresponds to $True$, i.e. *success* in the trivial case where there are no variables left to assign.

If you instead propagate the `6` first then the result is:

```
[[-7]]
```

and further propagation of `-7` will again lead to `[]`, i.e. *success*.

Whenever the current CNF becomes `[]` (*success*) the result returned is the *singleton* list comprising the list of all unit clauses that were propagated during the current invocation of the algorithm; in this case this is:

```
[[2,3,-7,6]]
```

which corresponds to the (single) assignment $b \to True, c \to True, g \to False, f \to True$. The order of the elements in the list is unimportant.

The second invocation of the algorithm proceeds similarly, this time by adding the unit clause `-2` to the above CNF, e.g. via `[[-2],[2,5],[3,-2],[3,-5],[-3,-7],[7,6],[-3,6]]`, before recursing. This corresponds to setting b to $False$. Proceeding similarly to the above, successive propagations again reduce the CNF to `[]` (*success*) and the following (single) assignment is returned:

```
[[-2,5,3,-7,6]]
```

To generate the result list of satisfying assignments in the top-level call to DP, all that remains is to concatenate the list of unit clauses that were propagated prior to the recursive calls, i.e. `4` and `-1` in the example, to each assignment returned from those calls. This yields the following:

```
[[4,-1,2,3,-7,6],[4,-1,-2,5,3,-7,6]]
```

Notice that in the first assignment there is no reference to variable number `5`, i.e. to e in the original formula. This means that the result implicitly encodes *two* solutions, `[5,4,-1,2,3,-7,6]` and `[-5,4,-1,2,3,-7,6]`, as the solution is insensitive to the value assigned to e.

If, at any point, one of the clauses in the flattened CNF becomes `[]`, e.g. `[[-3],[],[1,-2,4]]`, then the algorithm *fails* by returning the empty list of assignments, `[]`. This is because an empty disjunction corresponds to $False$ and a conjunction containing a $False$ term is itself $False$.

5 What to do

There are four parts to this test. The first three parts are independent, so if you get stuck on one part you may wish to move on to another. The module `TestData` contains a number of constants and functions that you can use for testing, including the formulae in Table 1 above. You should refer to the comments in `TestData.hs` for further details.

5.1 Part I: Utilities

1. Define a function `lookUp :: Eq a => a -> [(a, b)] -> b` that will look up an item in a list of (item, value) pairs. For example,

```
*SAT> lookUp 3 [(2,1),(3,8)]
8
```

   A precondition is that the item being looked up has a unique binding in the list.

   [1 Mark]

2. Using `nub` and `sort` from the imported module `Data.List`, define a function `vars :: Formula -> [Id]` that will return a sorted list of the variable names appearing in a propositional formula, without duplicates. For example,

```
*SAT> f1
Not (Not (Var "a"))
*SAT> vars f1
["a"]
*SAT> printF cnf3
((!c | !g) & ((b | (c | e)) & ((a | !b) & ((a | !e) & ((!a | !d) &
((c | (d | f)) & ((!a | !f) & g)))))))
*SAT> vars cnf3
["a","b","c","d","e","f","g"]
```

   Note that the function can be applied to any formula, including those in CNF.

   [3 Marks]

3. Using `vars`, define a function `idMap :: Formula -> IdMap` that will generate a list of `(Id, Int)` pairs associating the variable identifiers in a given formula with the integers 1, 2, ... and so on, in order. For example,

```
*SAT> idMap cnf3
[("a",1),("b",2),("c",3),("d",4),("e",5),("f",6),("g",7)]
```

   Note that the elements are implicitly in sort order by virtue of `vars`.

   [1 Mark]

5.2 Part II: Normal forms

1. Define a function `toNNF :: Formula -> NNF` that will convert a given formula into Negation Normal Form (NNF), as per Section 2.1.1. For example:

```
*SAT> toNNF f1
Var "a"
*SAT>  printF f6
!(b | (a & !c))
*SAT> printF (toNNF f6)
(!b & (!a | c))
```

```
*SAT> printF f8
((a & !(b | c)) | !d)
*SAT> printF (toNNF f8)
((a & (!b & !c)) | !d)
```

[4 Marks]

2. Define a function `toCNF :: Formula -> CNF` that will convert a given formula into Conjunctive Normal Form (CNF). In order to do this your first need to convert it to NNF, using `toNNF`, and then convert the result into CNF using the distribution rules given in Section 2.1.2. These rules are predefined in the template in the form of a function `distribute :: CNF -> CNF -> CNF`. Your `toCNF` function should thus use a helper function to recurse into the arguments of all `And`s and `Or`s, replacing each `Or` with a call to `distribute`. Note that all `Not`s will be applied to `Var`s only, as the formula is in CNF, and therefore also in NNF. For example:

```
*SAT> printF f4
!(!a & !b)
*SAT> printF (toCNF f4)
(a | b)
*SAT> printF f6
!(b | (a & !c))
*SAT> printF (toCNF f6)
(!b & (!a | c))
*SAT> printF f8
((a & !(b | c)) | !d)
*SAT> printF (toCNF f8)
((a | !d) & ((!b | !d) & (!c | !d)))
```

[3 Marks]

3. Using `idMap`, define a function `flatten :: CNF -> CNFRep` that will generate the flattened version of a given CNF formula. For example,

```
*SAT> printF c3
(a & b)
*SAT> flatten c3
[[1],[2]]
*SAT> printF c4
(a | b)
*SAT> flatten c4
[[1,2]]
*SAT> printF cnf3
((!c | !g) & ((b | (c | e)) & ((a | !b) & ((a | !e) & ((!a | !d) &
((c | (d | f)) & ((!a | !f) & g)))))))
*SAT> flatten cnf3
[[-3,-7],[2,3,5],[1,-2],[1,-5],[-1,-4],[3,4,6],[-1,-6],[7]]
```

The order of the elements in the resulting list, and their sublists, is unimportant. Note that `idMap` is defined to take a `Formula`, rather than a `CNF`. However, the types `CNF` and `Formula` are synonymous, as all CNFs are valid formulae.

[4 Marks]

5.3 Part III: The DP algorithm

1. By implementing the rules described in Section 4.1, and illustrated in Section 4.2, define a function `propUnits :: CNFRep -> (CNFRep, [Int])` that will take the flattened version of a CNF and return a pair comprising:

   - The `CNFRep` that results from propagating all unit clauses in the original `CNFRep`
   - The list of all unit clauses propagated as part of this process

   By definition, the `CNFRep` returned in the result pair should contain no unit clauses.

   Note that a unit clause in a `CNFRep` is a singleton list comprising just an integer, which may be positive or negative, depending on whether or not the corresponding variable is negated, as described in Section 2.2. For example:

   ```
   *SAT> propUnits []
   ([],[])
   *SAT> cnf1Rep
   [[1,2,3],[1,2,-3],[1,-2,3],[1,-2,-3],[-1,2,3],[-1,2,-3],[-1,-2,3]]
   *SAT> propUnits cnf1Rep
   ([[1,2,3],[1,2,-3],[1,-2,3],[1,-2,-3],[-1,2,3],[-1,2,-3],[-1,-2,3]],[])
   *SAT> cnf2Rep
   [[-1,-4],[2,1,5],[3,-2],[4],[3,-5],[-3,-7],[1,7,6],[-3,6]]
   *SAT> propUnits cnf2Rep
   ([[2,5],[3,-2],[3,-5],[-3,-7],[7,6],[-3,6]],[4,-1])
   ```

   The order of the elements in the resulting lists and sublists is unimportant.

   Hint: It's easier if you propagate one unit clause at a time, i.e. if a CNF has multiple unit clauses then select one of them to propagate. The others will be picked up by recursion.

   [5 Marks]

2. To complete the DP algorithm, use the function `propUnits` to define a function `dp :: CNFRep -> [[Int]]` that will return the list of all satisfying assignments for the given `CNFRep`, as described in Section 4.

   For example,

   ```
   *SAT> dp cnf1Rep
   [[1,2,3]]
   *SAT> dp cnf2Rep
   [[4,-1,2,3,-7,6],[4,-1,-2,5,3,-7,6]]
   *SAT> dp cnf3Rep
   []
   ```

   The order of the elements in the resulting lists and sublists is unimportant. Note that `cnf1Rep` has only a single satisfying assignment and `cnfRep3` has none. As described in Section 4.2, `cnf2Rep` has three solutions, but two of them are implicitly encoded in the assignment `[4,-1,2,3,-7,6]` where the variable corresponding to the number `5`, i.e. e here, can assume either the value $False$ or $True$.

Hint: The termination conditions (Section 4.2) can be based on inspection of the CNF *after* unit propagation via `propUnits`, noting, as in the example above, that if the given CNF is `[]` then the CNF in the pair returned by `propUnits` will also be `[]`.

n-Queens: The `TestData` module contains a function that will build a propositional formula representing the constraints that must be satisfied in order to solve the n-queens problem. If you get your `dp` function to work correctly, you might like to time it on the 8-queens problem; you can get GHCi to time executions by typing `:set +s` at the prompt. There's also a function `printBoard` in `TestData` that will print the chess board given a single satisfying assignment (a `CNFRep`) and the size of the board (an `Int`); you can use this for testing. Thus, for example:

```
*SAT> let sols = dp (queensCNFRep 8)
*SAT> length sols
92
*SAT> printBoard (head sols) 8
O O O O O O O X
O O O X O O O O
X O O O O O O O
O O X O O O O O
O O O O O X O O
O X O O O O O O
O O O O O O X O
O O O O X O O O
```

Your own solver may produce solutions in a different order, so your first solution may be different, but it should be a valid solution and there should be 92 of them for the 8-queens problem.

(Unassessed: How long does it take for $n = 8, 9, 10, 11...$?)

[4 Marks]

5.4 Part IV: All-SAT (Bonus question)

Finally, using the utility functions from Part I, or otherwise, define a function `allSat :: Formula -> [[(Id, Bool)]]` that will return the list of all satisfying assignments for a given propositional formula as a list of `(Id, Bool)` pairs, ordered by `Id`. The formula cannot be assumed to be in conjunctive normal form so you will have convert and flatten it before invoking `dp`. The post-processing of the result should produce a *complete* set of variable assignments, including the $False$ and $True$ bindings of any variables missing in an assignment returned by `dp` (see Section 4.2 above). For example:

```
*SAT> allSat cnf1
[[("a",True),("b",True),("c",True)]]
*SAT> allSat cnf2
[[("a",False),("b",True),("c",True),("d",True),("e",False),("f",True),("g",False)],
 [("a",False),("b",True),("c",True),("d",True),("e",True),("f",True),("g",False)],
 [("a",False),("b",False),("c",True),("d",True),("e",True),("f",True),("g",False)]]
*SAT> allSat cnf3
[]
```

```
*SAT> allSat cnf4
[[("a",False),("b",True),("c",True),("d",False)],
 [("a",True),("b",False),("c",False),("d",True)]]
```

Note that `cnf4` is a variant of the n-queens problem with rooks instead of queens, assuming a 2×2 board. `a` and `b` correspond to the top row and `c` and `d` the bottom row.

[Bonus 2 Marks]

Good luck!