As described in lecture 9, implement either the DP or the DPLL algorithm in ML. Your code should include a function called `dp` or `dpll` that takes a single argument: a list of integers representing a SAT problem in the DIMACS format. Your function should return `true` if the formula is satisfiable and `false` if the formula is not satisfiable.

Examples:
- `dp [1, 0, ~1, 0];`
  > val it = false : bool
- `dp [1, 2, 0, ~1, ~2, 0];`
  > val it = true : bool
- `dpll [~1,~3,0,~1,~5,0,~3,~5,0,~2,~4,0,~2,~6,0,~4,~6,0,1,2,0,3,4,0,5,6,0];`
  > val it = false : bool

Here are some suggestions to get you started:

- Take a look at the ML code posted on the class web site. The file “lib.sml” contains some useful helper routines that you may want to use. The file “sat.sml” contains an ML implementation of the SAT algorithm used in the Scheme assignment. The other files contain more examples of problems. Some of them are very hard. They are only there in case you want to try a variety of hard problems. I don’t necessarily expect you to be able to solve all of them. The simplest ones are “hole2”, “hole2_sat”, and “notprime2”.

- Recall that a unit clause is a clause with only a single literal. Write a function `getUnitLit` that takes a single argument, a list of clauses, and returns 0 if there are no unit clauses. Otherwise, if there is at least one unit clause, it returns a literal (non-zero integer) that appears in a unit clause.

  Examples:
  - `getUnitLit [[1, 2], [~1, 2], [3], [3, 1]];`
    > val it = 3 : int
  - `getUnitLit [[1, 2], [~1, ~2]];`
    > val it = 0 : int

- Write a function `unitProp` that takes two arguments: a literal and a list of clauses. It returns a new list of clauses resulting from doing unit propagation with the given literal. Recall that unit propagation with a literal $l$ removes all clauses containing $l$. It also removes the negation of $l$ from those clauses that contain the negation of $l$. You may find the helper functions “mem” and “del” to be useful.
Examples:

- `unitProp 3 [[1, 2], [~1, ~3], [3], [3, 1]]`;
  > `val it = [[1, 2], [~1]] : int list list`
- `unitProp ~1 [[1, 2], [~1, ~2]]`;
  > `val it = [[2]] : int list list`
- `unitProp 1 [[1], [~1]]`;
  > `val it = [[]] : int list list`

- Write a function `getPosNeg` that takes a list \( l \) of clauses and returns a pair of lists: the first list in the pair should contain all the variables that appear *positively* in \( l \) and the second list in the pair should contain all the variables that appear *negatively* in \( l \). You may find “mem” and “union” to be useful helper functions.

- Write a function `getPureLit` that takes a list \( l \) of clauses and returns 0 if it does not contain any pure literals. Otherwise, it returns a literal: either a variable that appears only positively in \( l \) or the negation of a variable that appears only negatively in \( l \).

- Write a function `pureLitRule` that takes two arguments: a literal and a list of clauses. It returns a new list of clauses resulting from applying the pure literal rule with the given literal. Recall that the pure literal rule removes all clauses in which a pure literal appears.

Examples:

- `getPosNeg [[1, 2], [~1, 3], [~1, ~2]]`;
  > `val it = ([1, 2, 3], [1, 2]) : int list * int list`
- `getPureLit [[1, 2], [~1, 3], [~1, ~2]]`;
  > `val it = 3 : int`
- `pureLitRule 3 [[1, 2], [~1, 3], [~1, ~2]]`;
  > `val it = [[1, 2], [~1, ~2]] : int list list`

- If you choose to implement the DP algorithm, write a function implementing the resolution rule. Note that the resolution rule may create a clause that contains both a variable and its negation. These clauses are automatically true and should be filtered out. In order to apply resolution, you must pick a particular variable to use for the rule. Think about how you should decide which variable to pick.

Examples:

- `resRule [[1, 2], [~1, 3], [~2, ~3]]`;
  > `val it = [[1, 2], [~1, ~2]] : int list list`
- `resRule [[1, 2], [~1, ~2]]`;
  > `val it = [] : int list list`

- If you choose to implement the DPLL algorithm, you will have to implement a mechanism for trying two different cases recursively. You will have to pick a variable to split on. Again, think about how you should pick the variable.

- Finally, write a function that checks the following cases:

  1. If the set of clauses is empty, it returns `true`
  2. Otherwise, if the set of clauses contains an empty clause, it returns `false`
  3. Otherwise, if there is a unit clause, it applies unit propagation and then calls itself recursively
  4. Otherwise, if there is a pure literal, it applies the pure literal rule and then calls itself recursively
  5. Otherwise, it applies either the resolution rule or the splitting rule and calls itself recursively

  - The function `dp` or `dpll` should call `getClauses` and then call the function described above.

For extra credit, implement both DP and DPLL and compare their performance.