Assignment 4

Part 1: A Simple Knowledge Base

Your task in this assignment is to implement and test a **definite clause theorem prover**. Definite clauses are a subset of the regular propositional calculus that, while not as expressive as full propositional logic, they can be used to do very efficient reasoning.

Definition of Definite Clauses

A **definite clause**, or **rule** for short, is a propositional logic expression of this form:

```
head <-- b_1 & b_2 & b_3 ... & b_n
```

`head`, and `b_1` to `b_n` are all ordinary **propositional logic variables** that are either true or false. We will usually call these **atoms** for short. The precise definition of legal atoms names is given in the function `is_atom(s)` given below.

`n` is an integer greater than, or equal to, 1. There must be exactly one atom, `head`, on the left of the `<--`.

`&` is logical "and", i.e. the ordinary and operator from propositional logic.

`<--` is the reverse of the usual propositional logic implication operator, i.e. `X <-- Y` means "Y implies X".

The rule above cam be read "if `b_1` and `b_2` … and `b_n` are all true, then `head` is true``. Or, alternatively, "for `head` to be true, `b_1` and `b_2` … and `b_n` must all be true". We will sometimes informally uses the word "prove" instead of "true" here, and say things like "to prove `head` is true, you must prove `b_1` and `b_2` … and `b_n` are all true".

Note that there is no "not" or "or" in these rules, and so it is not possible to represent all propositional logic sentences as definite clauses.

Definition of a Knowledge Base File

For this assignment, a **knowledge base file** (**KB file** for short) consists of 1, or more, rules (as described above). For simplicity, we require that each rule be written on its own line. Blank spaces are permitted between rules, and extra whitespace is permitted around tokens.

So, for instance, here is a knowledge base file consisting of three rules:

```
snowing <-- cloudy & below_zero & white_stuff_falling

cloudy <-- no_sun & daytime

below_zero <-- very_cold
```

Logically speaking, the order in which we write the rules doesn't matter, although you should try to group related ones together to increase readability.

For simplicity, you can assume that all rules in a KB file have a *different* head atom. In other words, you can assume that you will never have a situation like this:

```
cloudy <-- no_sun & daytime

cloudy <-- hard_to_see
```

Note

In this assignment, be careful how you use the words “true” and “false”. Informally, if an atom is in the KB then we can say it’s “true”, and if it’s not in the KB we can say it’s “false”. But just because an atom is in our KB doesn’t mean it is **in fact** true, and just because it is not in our KB doesn’t mean it is **in fact** false. For instance, a KB could contain the fact “its_raining” when it is not actually raining. It is more accurate to say the KB *believes* it is raining.

So, because of these concerns, it’s a good idea for this assignment to use the terms “in” and “out” instead of “true” or “false”

Definition of a Variable

The following function gives the precise definition of what strings can be used as atoms (i.e. variables) in definite clauses:

```
# returns True if, and only if, string s is a valid variable name
def is_atom(s):
    if not isinstance(s, str):
        return False
    if s == "":
        return False
    return is_letter(s[0]) and all(is_letter(c) or c.isdigit() for c in s[1:])

def is_letter(s):
    return len(s) == 1 and s.lower() in "_abcdefghijklmnopqrstuvwxyz"
```

Question 1: Interactive Interpreter

Implement an interpreter that lets a user interact with the KB. When run, your interpreter should display a prompt, e.g.:

```
kb>
```

It waits until the user enters a command (see below), and then tries to perform the command. If the user enters an invalid command, or the command cannot be run, then an error should be printed, e.g.:

```
kb> flurb
Error: unknown command "flurb"
```

Your interpreter should *never crash*: any error should just cause a helpful error message to be printed.

Your interpreter must implement (at least) these commands:

- `load someKB.txt`: This loads into memory the KB stored in the file `someKB.txt`; of course, you can replace the name `someKB.txt` with whatever the name of the KB file is. For example:

  ```
  kb> load sample1.txt
     snowing <-- cloudy & below_zero & white_stuff_falling
     cloudy <-- no_sun & daytime
     below_zero <-- very_cold

     3 new rule(s) added
  ```

  If a KB file already happens to have been loaded, then calling `load` again will delete the old one and replace it with the new one.

  If you try to load an incorrectly formatted KB file, then an error is printed. For example:

```
kb> load sample2.txt
Error: sample2.txt is not a valid knowledge base
```

- `tell atom_1 atom_2 ... atom_n`: This adds the atoms `atom_1` to `atom_n` to the current KB. For example:

  ```
  kb> tell a b c d
    "a" added to KB
    "b" added to KB
    "c" added to KB
    "d" added to KB
  ```

  If some `atom_i` is invalid (according to `is_atom`), then don't add *any* variables and print an error message, e.g.:

  ```
  kb> tell a b 4c d
  Error: "4c" is not a valid atom
  ```

  `n` is an integer greater than 1. It's an error to type just "tell", e.g.:

  ```
  kb> tell
  Error: tell needs at least one atom
  ```

  If you `tell` an atom that is already in the KB (due to a previous `tell` command, or because it was inferred by the rules), then `tell` should say something like "atom X already known to be true". For example:

  ```
  kb> tell sunny
    "sunny" added to KB

  kb> tell sunny
    atom "sunny" already known to be true
  ```

- `infer_all`: Prints all the atoms that can currently be inferred by the rules in the KB. Note that no atoms can be inferred until at least one `tell` command is called.

  When `infer_all` finishes, print all the inferred atoms, and clearly label them as inferred. Also, list and clearly label all the atoms that were known to be true *before* calling `infer_all`.

  For example:

  ```
  kb> load sample1.txt
    3 definite clauses read in:
     snowing <-- cloudy & below_zero & white_stuff_falling
     cloudy <-- no_sun & daytime
     below_zero <-- very_cold

  kb> tell no_sun
    "no_sun" added to KB

  kb> infer_all
    Newly inferred atoms:
       <none>
    Atoms already known to be true:
       no_sun

  kb> tell daytime
    "daytime" added to KB

  kb> infer_all
    Newly inferred atoms:
       cloudy
  ```

```
  Atoms already known to be true:
     no_sun, daytime

kb> infer_all
  Newly inferred atoms:
     <none>
  Atoms already known to be true:
     cloudy, no_sun, daytime

kb> tell very_cold white_stuff_falling
  "very_cold" added to KB
  "white_stuff_falling" added to KB

kb> infer_all
  Newly inferred atoms:
     below_zero snowing
  Atoms already known to be true:
     cloudy, daytime, no_sun, very_cold, white_stuff_falling
```

Note that you can add other commands if you like. For instance, a command like "clear_atoms" that removes all atoms might be useful for debugging.

Here is pseudocode for `infer_all` that you should follow:

```
Algorithm infer_all(KB)
  Input : KB, containing rules and atoms
  Output: set of all atoms that are logical consequences
          of KB

  set C to {}    // C contains the inferred atoms

  loop until no more rules from KB can be selected:
     select a rule "h <-- a_1 & a_2 & ... & a_n" from KB
            where all a_i are in C or KB (for 1 <= i <= n)
                  and h is not in C or KB
     add atom h to C

  return C
```

Question 2: Your Own KB

Design an interesting and non-trivial KB for some domain of interest to you. The rules and atoms are up to you. Try to make the KB as useful as possible, ideally usable for some practical application.

Put this KB in a file named `a4_q2_kb.txt` so that the marker can test it. Also, include a file named `a4_q2_readme.txt` that includes an explanation of your KB (i.e. how did you come up with the rules?), and gives examples of how to use it.

Part 2: Project Preparation

Assignment 5 is a project that you can work on either by yourself, or with a partner (teams larger than 2 are not permitted).

So, in a file named `project_info.txt`, tell us this information:

- The name of everyone on your team. Teams may have 1 or 2 people (teams larger than 2 are not permitted).

  If you are on a team of 2, you will need to talk to your TA about signing up on a team in Canvas.

  We expect a more work from teams of 2, and we expect both members to do an equal amount of work. Teams of 2 that don't follow these rules may have their marks adjusted appropriately.

- The project (see below) that you've chosen to work on.

- The main programming language you plan to use, plus a list of any libraries you plan to use. You can use any language that the TAs can run on their computer — if you are not sure, then please email the TAs to check!

  Please note that general-purpose helper libraries of code *are permitted*, but no libraries that implement key AI techniques are permitted: the purpose of this assignment is for you to implement some AI algorithms on your own, and so calling existing versions of the algorithms is not what we want to see.

Project: Sokoban

Sokoban is a classic box-pushing puzzle played on a grid of cells. You play a warehouse worker who can push (but not pull) one crate at a time. The goal is to push all the crates onto specific locations marked on the floor. Some Sokoban puzzles are easy while some, even small ones, can be surprisingly challenging.

Please take a few minutes to play Sokoban online (such as here), or with a free mobile app, to understand how it works.

In this project, your task is to find optimal solutions to Sokoban puzzles. The markers will be looking for the following:

- Use of a memory-efficient A* algorithm, such as IDA*. Please use a language other than Python, and make your implementation as efficient as possible.

  If you get help, or ideas, or bits of code from elsewhere, tell us where! Cite all the help you used to make this algorithm.
- Use of the basic Sokoban input format specified here. Your solver should be able to read any puzzle specified in this format.
- Solutions are in the LURD format. Capital letters are used to indicate a move that pushes a box.
- Your solver can recognize unsolvable puzzles.
- You list solutions to puzzles from some known collection of Sokoban puzzles (you can easily find collections of puzzles on the web). While you can certainly make up some of your own puzzles, it is important to demonstrate that your solver is correct and can solve well-known puzzles.
- For small/easy puzzles, your solver should be efficient. Sokoban is PSPACE-complete, which means there is likely no efficient general-purpose solver.
- You've put some effort into finding good heuristics, using pattern databases, or something beyond basic IDA* search.
- You clearly specify the **biggest and most complex** problem your solver can solve in at most *5 minutes*.
- You clearly specify the **smallest** problem that your solver **cannot** solve in *5 minutes* or less. Please make a non-trivial effort to find this smallest problem. For example, you could write a program that generates Sokoban puzzles and then solves them. Keep track of the solving time and the solution length.

  Suggestion: To estimate the complexity of a *solution*, you could use the formula $\frac{B \cdot P}{E}$; B is the number of box moves in the solution, P is the total number of pusher moves, and C is the number of empty cells in the puzzle. Find a puzzle where E is as small as possible, and the $\frac{B \cdot P}{E}$ value of an optimal solution is as big as possible.
- A written report of at least one page that explains your program, the techniques it uses, its results, etc. You could include graphs and tables of relevant data.

Spelling, grammar, and formatting matter.

Project: Fast Implementations of WalkSAT and Resolution Proving

Implement an efficient version of *both* WalkSAT, and a propositional theorem prover based on resolution. They should both use the minisat input format. Using the same set of test examples, compare the performance of your two algorithms against the Python version of WalkSAT (in the textbook code in `logic.py`) and minisat. Use randomly generated k-CNF sentences as input (these are described in the textbook in the first full paragraph of page 264).

The markers will be looking for the following:

- Implementation of WalkSAT using an efficient programming language, such as C++ (Python is probably not efficient enough).
- Implementation of a propositional theorem prover based on resolution using an efficient programming language, such as C++ (Python is probably not efficient enough).
- Speedy solving times: not only should you use an efficient programming language, but you should use efficient data structures and algorithms, plus anything else you can think of to make this run efficiently.
- Randomly generated k-CNF expressions. Note that *this* part of your project could probably be done in Python, as efficiency is not as important as it is for the main solvers.
- A written report of at least one page that explains your programs, the techniques they use, the results of your comparisons, etc. This should include graphs and tables of relevant data.

Project: Reversi with Monte-Carlo Tree Search

Re-do assignment 3, but instead of Tic Tac Toe use Reversi (on an 8-by-8 board).

Make your program play as well as it can. It doesn't need to play perfectly — Reversi is a much harder game than tic tac toe.

The markers will be looking for the following:

- Use of a language more efficient than Python. Doing lots of playouts as quickly as possible is the key to success. You should keep an estimate of how many playouts per second your program can do as a measure of its performance.
- You program should play as well as possible.

  So that games don't go on too long, you should put a limit on the maximum amount of time the computer can use (e.g. 5 seconds at most per move).
- In addition to the pure Monte Carlo Tree Search version of your program, create a second modified version of your program that uses heuristics — and any other additions you think would help — to make (hopefully) better-than-random moves during the playouts.

  Make this version of your program play against the other version to see which one is better.
- A written report of at least one page that explains your implementations, discusses how well they play, the extra techniques used by your second version, a comparison of which program is stronger, etc. This should include graphs and tables of relevant data.

What to Submit

For this assignment you should submit the following in a zip file named `a4.zip`:

- `a4.py`, containing all the code for the marker to run and test question 1

- `a4_q2_kb.txt`, containing all the rules for your own KB
- `a4_q2_readme.txt`, containing an explanation of `a4_q2_kb.txt`, and examples of using it
- `project_info.txt`, which contains the requested information about your project for assignment 5

Use the exact file and function names — otherwise marking scripts might give you 0!

Don't use any code from the textbook, or any special libraries for this: stick to regular Python, and code imported from its standard library.