PROJECT SUMMARY: Robbie's "blocks world" starts in an INITIAL STATE (i.e., a particular configuration of the blocks and Robbie's hand). From there, Robbie moves blocks around in response to user requests to do a series of specific MOVEs, resulting in a new CURRENT STATE after each move is done. Note that the user might ask Robbie to do something that can't be done. Robbie can also display a simple "graphical" picture of the current state of the world, when requested. Upon termination of a session, Robbie saves the current state to a file so that a subsequent session can start with either the initial state or with the current (saved) state, depending on what the user specifies.

ROBBIE'S RESPONSES: Robbie ALWAYS RESPONDS to every user request
- for a `showWorld` request, Robbie shows a graphical picture of the current world
- for any SUCCESSFUL MOVE request, Robbie says:

OK, I did a _____

[specifying what he did just do – i.e., the move and the block(s)/things involved]

- for any UNSUCCESSFUL MOVE request, Robbie says:

SORRY, I can't _____

[specifying what he can't do – i.e., the requested move and the block(s)/things involved]

[Note: use these exact 2 responses, including CAPS to make it easier to grade].

Do only ONE RESPONSE PER REQUEST, not one response for each partial-action!

THE CONTROLLER: At this stage, Robbie does NOT derive his own PLANS to transform the world from the current state to some desired goal state. Rather, a human user derives a PLAN and enters a series of requests (moves) to lead Robbie to achieve the desired state.

Robbie can respond to requests entered by the user ("interactive mode") or from a Prolog driver with the specific "plan" (i.e., "batch mode"). *(Note: The MOVE rule names must match my specs exactly (including shortcuts) since that's what my `thePlan` rule requests use).*

SOME GENERAL ASSUMPTIONS:
- There's enough room on the table to hold all blocks in this blocks world, even if they were all actually ON the table all at once.
- There is room in the world vertically to stack ALL blocks on top of one another, if requested.
- Left and right refer to the audience-view (not Robbie-view).
- Robbie might be asked to do something he can NOT do, because it's either:
  - IMPOSSIBLE in his world: e.g.,
    - moving himself or the table
    - putting something on himself
    - dealing with a non-existent object
    - etc.
  - NOT POSSIBLE NOW given the current state & the requested move: e.g.,
    - he's already holding something when he's asked to . . .
    - some thing's not `clearedOff` when he's asked to . . .
    - some thing's the wrong shape to have something on it
    - he's holding the wrong thing when he's asked to . . .
    - etc.

$$$$$$$$$$$$$$$$$$$$$$$$$ **THE 9 CODE FILES** $$$$$$$$$$$$$$$$$$$$$$$$$

- 9 physically separate files
- use names specified below (or something similarly descriptive)
- don't change MY files' data (except if I made an error, in which case you MUST let me know) unless I specify that something needs expanding. For example, do NOT add:
  - facts for `leftmost` and `rightmost` since these can be inferred with rules
  - facts or rules just because you had them in asgn 1 (unless they're needed here)

`1 - staticDatabase` *[MY FILE]*
- description of the unchanging aspects of the world – stored as individual property facts (vs. record-based facts like A1) *(NOTE: most properties used in A1 not used in here to simplify A2)*

`2 - initialState` *[MY FILE]*
- dynamic DB - changeable aspects of the world including: `on left holding`
- add Prolog's "`dynamic`" directive predicates for these 3 (since SWI Prolog needs them)
- don't add additional predicates

`3 - currentState` *[YOUR FILE] [THIS FILE WON'T EXIST, INITIALLY]*
- similar to `initialState` file, except it'll be a different configuration of the blocks
- this file will be created by your program - see driver file description below
- if the program is run several times, then this file will be over-written each time

`4 - knowledgeBase` *[NOT USED IN ASGN 2]*
- not used in this project since all MOVE requests designate objects/things by NAME

`5 - moves` *[YOUR FILE]*
- use the predicates specified below since my `thePlan` rule will call these specific names
- your 8 move rules all have preconditions (1st) followed by actions (2nd)
  (it's important that ALL preconditions are checked before ANY actions are done)
- to help readability and to simplify the actual move rules themselves,
  add support rules (where needed) to this file (if they're called by the move rules)

`6 - shortCuts` *[MY FILE]*
- shorter names for the moves for easier interactive testing
- *(Note: if a German or. . . user interface were needed, you'd just need to add a simple file like this where you had rules to equate German or … words for pickup, putdown, etc .)*

`7 - showWorld` *[YOUR FILE – use MY FILE as a starting point]*
- use the predicates specified below since my `thePlan` rule (and the shortCuts) will call these specified names
- expand the rules I've provided into full Prolog rules
- add support rules to this file (where needed) if they're called by the `showWorld` rule
- this file also includes the `cheapShowWorld` rule which you'll use until you get the actual full `showWorld` rule finished

8 – driver ***[YOUR FILE]***

- contains the big `go` rule (i.e., "`main`") which does the following (and other things):
  - consult the appropriate files (except not `thePlan` or the `?State` files yet)
  - find out from the user whether they want batch mode or interactive mode and store that information for use later
  - call `getCorrectState` (in `stateStuff` code file) which determines & loads the correct starting State
  - if interactive mode specified:
    - give user a welcome message indicating what moves they can request (by specifying the list of 10 shortcut rules and what they stand for) including `sw` and `csw` (and quit)
    - loop: get a user request, then carry it out
    - quit → do `saveState` (in `stateStuff` code file) before halting
  - if batch mode specified:
    - consult `thePlan` file
    - direct output to a file instead of the screen ('`A2LogFile.txt`'), (& tell user the name of the file, before redirecting)
    - run `thePlan`
    - redirect output back to the screen
    - do `saveState` (in `stateStuff` file)
    - give user an ending message
- program should be self-starting – i.e., a directive which calls the big `go` rule.

9 – stateStuff ***[YOUR FILE]***

- includes 2 rules (and possibly some support rules which these 2 use) which `driver` calls:

1. `getCorrectState`
   - check which state user wants to start with: initial state or current state (the most recently saved state
   - make sure the proper one gets loaded
   - if user requests the most recently saved state, but `currentState` file does NOT exist, just automatically use `initialState` file instead (and let the user know you did that)
   - give a reassurance message to the user: '`??????? State Loaded`'
2. `saveState`
   - save the current state of the world to the `currentState` file *(yes, over-write the existing one, if any)*
   - this file only includes the predicates which describe the STATE: `on left holding` (and their `dynamic` directive commands)
   - give a reassurance message to the user: '`Current State Saved`'

10 – thePlan ***[MY FILE]***

- 3 versions of this file for more thorough testing. Only 1 is used for a particular run of the program at once. Make your own test file, adding requests as you gradually add additional moves and features to your program.
- A tiny version of the rule would look like:

```
thePlan :-
     showWorld,
     pickUp(a),
     pickUp(robbie),
     putDown(a),
     showWorld.
```

$$$$$$$$$$$$$$$$$$$$$$$$$$$$ **THE 8 MOVES** $$$$$$$$$$$$$$$$$$$$$$$$$$$$

includes 6 base rules (below), 2 synonym rules (below) & 2 support rules (see next section)

```
pickUp(Block) :-                          % you complete this
putDown(Block) :-                         % you complete this
putDownOnRight(Block) :-                  % done – it's just a synonym
        putDown(Block).
putDownOnLeft(Block) :-                   % you complete this
putOn(Block,Object) :-                    % you complete this
putOnCarefully(Block,Object) :-           % you complete this
putOnRight(Block) :-                      % done – it's just a synonym
        putOn(Block, table).
putOnLeft(Block) :-                       % you complete this
```

Some Assumptions about the Moves

- Robbie can NOT deal with blocks which don't exist
- Robbie can NOT lift himself or the table up
- Robbie can only grasp things if his hand is ALREADY empty
- Robbie can only deal with blocks which are ALREADY cleared off
  - unless he's already holding the proper block, in which case, proceed with the move (i.e., it's OK)
- if Robbie is asked to `putOn` a block on something, it's considered OK if he's EITHER ALREADY holding the correct block NOW, OR if he's NOT ALREADY holding it but his hand is empty and he can pick it up
  - this case applies to all versions of the `putOn` move: plain/left/right/carefully
- but Robbie can only `putDown` a block he's ALREADY holding NOW
- Robbie can place a block on the table or on another block but not on himself (there's ALWAYS ROOM on the table)
- Robbie can NOT place a block on top of a pyramid or ball shaped block unless he's requested to do it CAREFULLY
  - but just because he's asked to do it carefully, doesn't mean it's necessarily a pyramid or a ball
- Robbie can grasp any shape block, including pyramids and balls (and, of course, cubes)
- when a block is to be placed on the table, Robbie assumes it means **"on the RIGHT end"** (that's "audience right") unless the request specifically indicates LEFT
- Robbie always keeps the blocks on the table "LEFT-JUSTIFIED" (by doing a `slideLeft`, when appropriate)

`slideRight` AND `slideLeft` - 2 More Move Rules

- add 2 support rules to the `moves` file: `slideRight` and `slideLeft`
- the user doesn't generally request these (but they MAY) – rather, the other move rules do [NOTE: `thePlan` rule will call them just to test your code!]
- Robbie MUST call these at the appropriate time (within some of the 6 base move rules)
- these rules don't actually DO ANYTHING (i.e., they don't change the world) but they:
  - must EXIST as rules and
  - must be CALLED at the appropriate times as an ACTION in the appropriate move rules

- `slideRight` is called only when Robbie needs to set a block down onto the table on the LEFT end of the table
  - so it's called BEFORE actually putting the block down
- `slideLeft` is called whenever either:
  - a block is removed from ON the table itself (if that block was NOT currently the rightmost block)
  - OR when Robbie just put a block on the LEFT end of the table,
    - since he would have done a `slideRight` just BEFORE to setting that block down, he now has to `slideLeft` AFTER setting that block down
    - so it's called AFTER actually putting the block down
- Thus Robbie always keeps the blocks on the table "LEFT-JUSTIFIED"
  - since most of the time blocks are set down on the RIGHT end of the table

Other support rules

- you'll want to add these (and other) support rules to the `moves` file:

```
leftost(Block) :-
rightmost(Block) :-
clearedOff(Thing) :-
```

A NOTE on clearedOff

- `clearedOff` is just a true/false (Boolean) type of predicate that does no action, per se
  - it just checks IF a block is currently ALEADY cleared off NOW
- is the table `clearedOff`?
  - Linguistically (i.e., the "meaning of the word"), NO
  - But practically (i.e., in terms of this project) YES - since Robbie only really wants to know if he can put a block ON that thing or not. And we're assuming that there is ALWAYS ROOM on the table (though a `slideRight` or `slideLeft` may be needed)

A NOTE on WRITING MOVE RULES:

- Robbie checks ALL precondition checks before doing ANY aspect of the move. That way he doesn't have to UNDO things he's partially done.
  - Remember that retract does both "checking" and action"
- Can `putOn` be implemented (partially) as just a call to `pickUp`, then a call to `putDown`?
  - NO because then `putOn` would end up being (in effect)
    - 1st - preconditions for `pickUp`
    - 2nd - actions for `pickUp`
    - 3rd - preconditions for `putDown`
    - 4th - actions for `putDown`
  - There'd be a problem since #2 would have gotten done BEFORE Robbie's sure all of #3 was checked.

$$$$$$$$$$$$$$$$$$$$$ **THE PICTURE OF THE WORLD** $$$$$$$$$$$$$$$$$$$$$
(`showWorld`)

Here's a typical picture of the world after some blocks have been moved. All block names must be CAPITALIZED. The blocks' SHAPES must look like I've shown. There should be some space between blocks, but there should be NO "WRAP-AROUND" if all blocks were actually ON the table (when this is printed for the demo you hand in).

Use a fixed-width font (like Courier) so things line up (for the demo you hand in) !!!

```
==========< [C]

     /L\
     (J)
     [G]                   (N)
     [I]        [K]        [M]
[A]  [B]  /D\  /E\  [F]   [H]
-------------------------------------------------
          ||                              ||
          ||                              ||
```

Note that the world above takes 1 + 1 + Height + 3 + 1 lines to print out, that is:
1 line for Robbie's hand (and the block he's holding, if any)
1 blank line above all the blocks
Height number of lines (in this case 5) for the tallest stack in the current world
(which could be anywhere from 1 to 13)
3 lines for the table (including legs)
1 blank line before future responses

$$$$$$$$$$$$$$$$$$$$$$$$ **SOME GENERAL NOTES** $$$$$$$$$$$$$$$$$$$$$$$

1) `initialState` (and in the future, `currentState`) contains `on` facts (just the blocks that are actually directly ON the table) in **left-to-right order**. You can use this natural ordering when doing `showWorld` (in `bagof`). So you should also use **`asserta`** and **`assertz`** appropriately when setting down blocks on the left & right ends of the table, respectively.

2) `assert` and `retract` change the facts in memory, but they do NOT change what is in the original Prolog file(s). (Remember that you need a `dynamic` directive for any predicate which might change or be missing).

3) `retract` only removes a single fact - the 1st one that it unifies with.
- `abolish` and `retractall` both work for removing all designated facts.
- however, `abolish` also removes the matching `dynamic` directive predicate (BAD!!!) which `retractall` does not do.

4) Robbie ALWAYS RESPONDS to every USER request. He does NOT respond to:
- every call to a Prolog rule,
- nor to your rules' calls to support rules,
- nor to your checks to see if blocks are `clearedOff` and/or if his hand is empty,
- nor to YOUR calls to utilities like `slideLeft` & `slideRight`
  - though he WOULD respond if the user (or `thePlan`) called them
- etc.

Robbie's responses are "**user-feedback**", not "**programmer-feedback**" (i.e.,a debugging aid).