Assignment 2: Prolog Talking Box Assignment

Assigned problem: Ten-Questions

In this assignment, the task given was to design an AI that thinks of a question (Answerer) and interact with an user (Questioner). For ease of implementation, we simply choose the game first, but more advanced implementation of Adversarial Search can be done. Due to the fact that user will probably not guess it in 10 questions, I found it quite pointless to implement adversarial search and focus my time on building 2 version of this game:

1. CLI version – 10Q_CLI_v9.pl
2. Web app game version – 10Q_SERVER_v7.pl

The two version has a slight difference in boundary checking implementation due to practical purposes (mainly prolog http support API), but the core of the game (specified in the Mechanics Rules section and Q-Handler section, together with KB_facts.pl dependency, are the same).

There are also helper files KB_facts.pl and html_form_v3.pl and generated_dialog.docx. More intruction on how to run each of the file can be found in README.txt

There are 3 parts to this report:

1. Design of Knowledge Base (shorthanded as KB from here)
2. Summary of KB
3. Generated Dialog

As a last note, this project is purely built from scratch without referral to existing github. Examiner might wish to see the documentation in the code to verify this.

1. Design of KB

With the context given, we can decompose this problems into the following stages

1. Design a structure that holds the information about the game
2. Design the 3 main queries predicates: has/1, has/2 and is/1
3. Design a wrapper to check turn and check round
4. (optional) Beautified printing for console output as well as instructions for users.

It is easy to notice that point 2. will be dependent one point 1., therefore it is important to iteratively design the data structure of 1 and see what works best.

This portion of the report will walkthrough the thinking behind the design of this game through the different stages of design, with a focus on brevity and roles of each rule in order to concisely help the examiner to visualize the thought process. All predicates are *italicized* for ease of reading.

Stage 1: Data Structure

There are 3 main facts for the game

1. The round it is at (cannot exceed 5)
2. The turn the player is at (cannot exceed 10)
3. The chosen game at this round and its propery
4. The list of game

The first 3 bullet points are stored in predicates: *round/1, turn/1,chosen/1 & property/1*

Each of the game can be stored in predicate *game/2 - game(game_name, list_of_property)*.

The list of game can then be generated by built-in rule *setof(Game, X^game(Game,X),GameList).*

Then we can assert this list into KB by *assert(gameList(GameList)).*

To ensure each *game/2* are consistent, we define *keyword/1* to specify the attributes of each elements in the 2nd argument of *game/2*.

keyword([teamsize,numOfTeam,fieldtype,equipment,gameMode]).

We specify the dependency on *keyword/1* as *has/2* and *game/2* as any addition of new attributes means all game need to update on this attribute and *has/2* will need to be updated.

Stage 2: Design the core rules

We introduces two new rules: *init/0* and *startNewRound/0*

We delegate the role of each predicate as followed:

1. *init/0* – set up the round counter, generate the *gameList/1* (as discussed in Stage 1 section). That is sufficient for the initialization.
2. *startNewRound/0* – check for round number and terminate if it exceeds 5, round increment, reset *turn/1*, choose the game using *random_member/2*, assert *chosen/1* and *property/1*

We note that for *chosen/1, property/1, turn/1 and round/1*, they all have to be *abolish/1* and then a new assignment has to be *assert/1* into the KB in order to update their values. We will see this structure **[_predicate(X),abolish(_predicate/1),update_value/?,assert(_predicate(X))]** throughout to do this updating of value. *update_value/?* Will depends on the predicate we are looking at.

Once the round has been started, these rules should be implemented:

has/1: has(X) -- checking X against the property_list stored in property/1 by invoking member/2.

has/2: has(Keyword, Value) -- provide more interactivity we can be detailed about these 3 cases

- Keyword is wrong – print out supported keyword
- Keyword is right, Value is wrong – print something informative
- Keyword is right and Value is right – print correct status

has/2 will need to implement *nth0/3* to get the element with the correct index to the list stored in *property/1*. As long as we make sure the nth0 follows the order of attribute specified by keyword/1, it should work well. Detailed implementation is in the comments of the code, but generally the idea is to provide for the three user cases stated above.

is/1 : -- match against *chosen/1*

As such at this stage we will have the predicates summarized in the table below:

Predicate	**Role in the game**
round/1	Store the current value of the round – which round is it now?
turn/1	Store the current value of the turn – which turn is it now?
chosen/1	Store the value of the game chosen at the current round
property/1	Store the value of the chosen game's attributes in a list
game/2	Specify all the games' name and attributes
keyword/1	Specify what are the attributes for each game in a list
init/0	Start the entire app, do the necessary initialization
restart/0	Clean up, and invoke *init/0*
startNewRound/0	Check *round/1*, reinitialize *turn/1*, update *chosen/1* and *property/1* . Update is done by **[_predicate(X),abolish(_predicate/1), update_value/?, assert(_predicate(X))]**
has/1	check if arg is *member/2* of *property/1*
has/2	*has/1* but cater to 3 different user cases
is/1	*chosen/1*

Stage 3: The turn/1, round/1 wrapper and other helpers

Now that the core logic of the game is in place, we need the *turn/1* and *round/1* to make it into a complete, functional game. This turns out to be much more difficult than it should.

Firstly, we need a counter for *turn/1* and *round/1* predicate to function. But as reassignment in a declarative language like Prolog is simply not allowed, we need to make use of *abolish/1* and *assert/1* to reupdate the value. In this case, we have chosen to use a list of n singleton variables to indicate the value of n. As such if you check turn(X) it will return a list of _G variables.

The helper function *len/2* was defined to return the length of a list via recursion:

len([],Res):- Res is 0.   -- If the list is empty, Result is 0

len([H|T],Res):-len(T,L), Res is L +1. – If the list is not empty, check the len L of the tail portion of the list, and Result will then by L+1

This is a simple recursion to check len(), easy to implement but expensive in computations.

Now turn/1 and round/1 (incrementing version) value update can be implemented by

Explanation	*turn/1*	*round/1*
Initialization	length(TurnCounter,1), assert(turn(TurnCounter)),	length(ROUND,1), assert(round(ROUND)),
Check the value	turn(T), len(T,N),	round(R), len(R,N),
Some boundary check	N==10 ->	N==6->….
Increment step	append(T,[_],NT),	append(R,[_],NR),
Reassignment	abolish(turn/1), assert(turn(NT))).	abolish(round/1), assert(round(NR)),

We can see that:

- Built-in *length/2* was used to create a list of length specified
- Built-in *append/3* was used to append a random _ into the list – thus having the effect of increase length of list by 1
- Predicate *len/2* was used to check length of the list
- *init/0* will implement the initialization for *round/1*
- *startNewRound/0* will implement the other 4 steps of *round/1*, as well as initialization of *turn/1*
- *checkTurn/0* will implement the other 4 steps of *turn/1*

We saw that checkTurn/0 predicate was created to check the current turn. The question then becomes where should checkTurn/0 be matched? For this assignment, it is decided that the rule is matched at the very last AND branch of *has/1*, *has/2* as well as the last AND part of the OR branch of *is/1.* Therefore, as it is a post-increment check, it only checks until N==10 – the terminating case. Consider this against round/1, we will check until N==6 as it is a pre-incremental check. This makes sense because we want to check the round before we carry it out, but for turn we want to check after the turn has ended (for ease of implementing in AND-OR tree -- we can also do pre-incremental check, but it simply takes more lines of code and very easily cause mismatched operator if the body of the code is very long – as we shall see in has/2 and startNewRound/0 implementation).

It can also be noticed that for the 5 steps of both turn/1 and round/1 value update cycle, the first stage is implemented by 1 rule, and the other 4 is implemented in another rule. This makes the code hard to trace, but easier to implement as there would be fewer boundary check this way.

Lastly, adept Prolog programmer will notice that startNewRound/0 will fail on the very first round in this implementation. This is due to turn/1 implementation, where we try to abolish(turn/1) in order to initialize before there was turn/1 in the KB. The easy way to salvage this is by having turn(0). in the KB, so we can abolish it on the very first round. This our code cleaner from having unnecessary condition checking.

A similar issue is faced with chosen/1 and property/1. We see that because startNewRound/0 has many roles to carry out in each round, and each role has a boundary check, thus it is the easy recipe for failure. Therefore, similarly, chosen(nothing). and property(nothing). was added for implementation purposes.

Stage 4: Beautify and interactivity with different user cases

After 6 iterations, the game has finally taken its most rudimentary shape. The logic has been implemented, with turn and round counter implemented to introduce the sequence to the game. In order to take this one step further, we will introduce printMenu/0 and other printing functions. We will explain the role of each printing below

Beauty printing	**Used in**	**Role in the game**
printmenu/0	printmenu/0	Print the introduction to the game, explaining the rules to users.
write(‘round ’), writeln(x)	startnewround/0	Print out round number
write(‘turn ’), writeln(x)	checkturn/0	Print out turn number
(N==10->(writeln('You ran out of turns. You lost.'), write('The chosen game was: '),writeln(X),writeln('End of Round'),	checkturn/0	Signify the boundary checking in checkTurn. Terminate the game’s round after this. Allow users to see which game they failed to guess. It is implied implicitly that once N==10 is assumed, the user hasfailed– due to the fact that is/1 will invoke startNewRound/0 if the decision made was correct, and hence reinitialize N = 1
member(X,L)-> (writeln('YES YOU ARE RIGHT'),checkTurn); write('No this game does not have '), write(X),	*has/1*	Response to the user’s queries has/1 using an OR branch.
(keyword(K),not(member(X,K))->(writeln('Supported keywords are: '),write(K), writeln('. You are still counted as wrong though.')));	2nd OR branch of *has/2*	Explain to user what are the supported keywords. Still penalize them nonetheless.
chosen(X)-> (writeln('Great Job! You guessed the game!'), startNewRound); write("Nice try but no it is not "),write(X), writeln(". Try again."), checkTurn.	*is/1*	If user guess correctly, print congratulatory messages, and startNewRound/0 – reset N == 1 and thus do not need to checkTurn OR print an error message and do a turn increment.

2. Summarised KB

After 10 iterations, the final KB version – with incrementing round counter and randomized new game for each round as well as keyword/1 finalized, is summarized in the table below. We also segregate the facts (all game/2 and keyword/1) into a separate file ('KB_facts.pl') and invoke it in the main file by :-['KB_facts.pl']

Main file:

Predicate	**Role in the game**
round/1	Store the current value of the round – which round is it now?
turn/1	Store the current value of the turn – which turn is it now?
chosen/1	Store the value of the game chosen at the current round
property/1	Store the value of the chosen game's attributes in a list
game/2	Specify all the games' name and attributes
gameList/1	Store the name of all the games in the KB
keyword/1	Specify what are the attributes for each game in a list
init/0	Start the entire app, do the necessary initialization
restart/0	Clean up, and invoke *init/0*
startNewRound/0	Check *round/1*, reinitialize *turn/1*, Use *delete/3* on *gameList/1* to delete the game from the previous round, Choose a random game using *random_member/2* on *gameList/1,* update *chosen/1* and *property/1* . Update is done by **[_predicate(X),abolish(_predicate/1), delete/3 assert(_predicate(X))]**
has/1	check if arg is *member/2* of *property/1*
has/2	*has/1* but cater to 3 different user cases
is/1	*chosen/1*
checkTurn/0	Check *turn/1*, perform boundary check, update its value as needed
len/2	Helper function to check the length of a list
chosen(nothing). *property(nothing).* *turn(0).*	Empty predicate for implementation purposes

Built-in predicates that are used: *length/3, abolish/1, assert/1, append/3, member/2, random_member/2, setof/3.*

KB_fact.pl

keyword([teamsize,numOfTeam,fieldtype,equipment,gameMode]).

/**dependencies: game/2 and has/2.**/

%The property arg of game (the 2nd one) must follow the order stated in keyword predicate%

game(archery, [1,many,field,bow,score]).

game(badminton, [2,2,court,racquet,score]).

game(basketball, [5,2,court,ball,score]).

game(volleyball, [6,2,court,ball,score]).

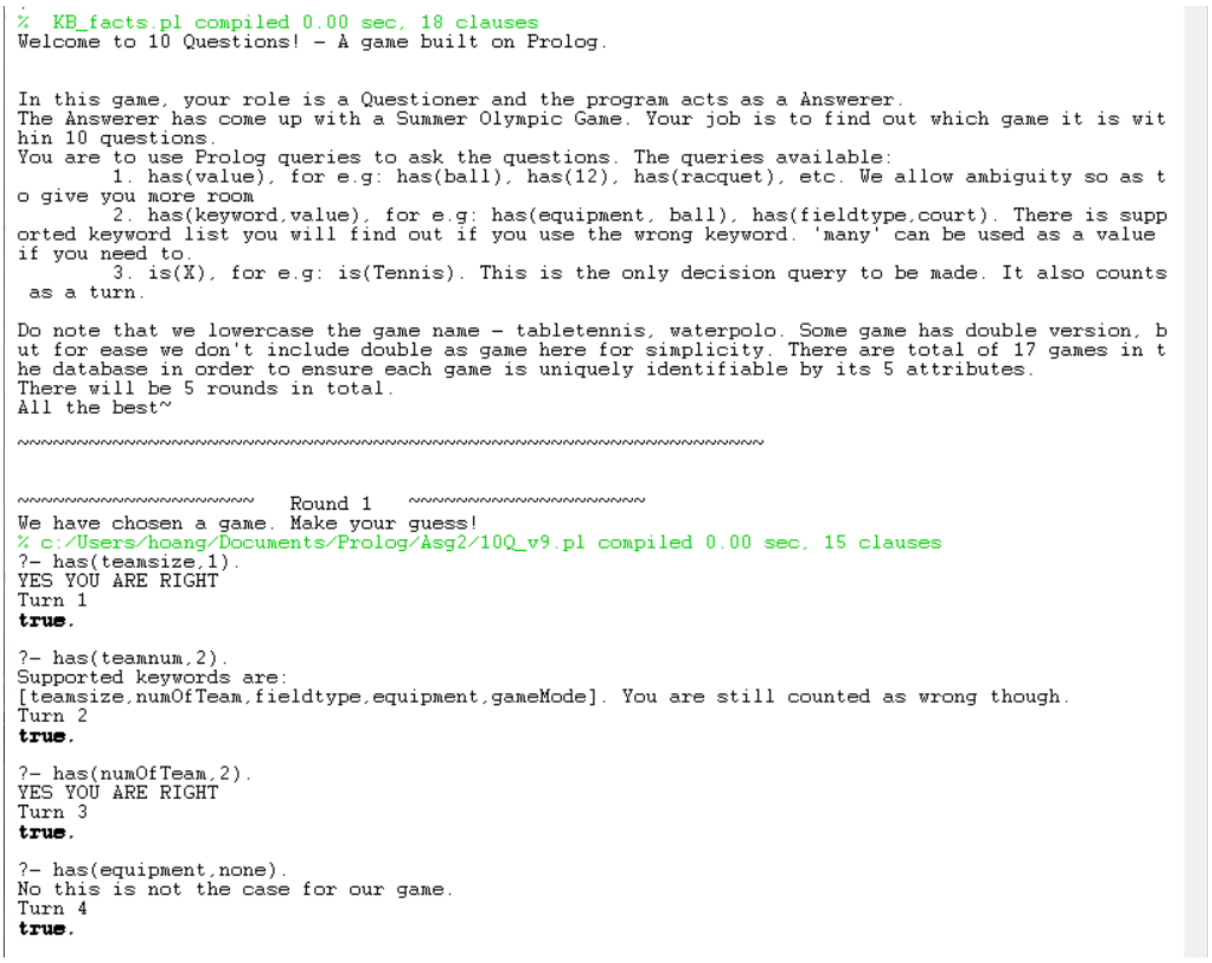
```
game(boxing, [1,2,ring,gloves, knockout]).                    %5th game
game(canoeing, [1,2, water, paddle,timed]).
game(cycling, [1,many,track,bicycle,timed]).
game(fencing, [1,2,ring,sword,knockout]).
game(hockey, [6,2,court,stick,score]).
game(handball, [7,2,court,ball,score]).               %10th game
game(judo, [1,2,ring,none,knockout]).
game(rugby, [15,2,field,ball,score]).
game(soccer, [11,2,field,ball,score]).
game(tabletennis, [1,2,table, bat,score]).
game(tennis, [1,2,field,racquet,score]).              %15th game
game(waterpolo, [7,2, pool, ball, score]).
game(weightlifting, [1,many, stage,barbell,score]).
```

3. Generated Dialog

Can also be found in generated_dialog.docx file submitted. I will only cover the CLI version in the interest of space.

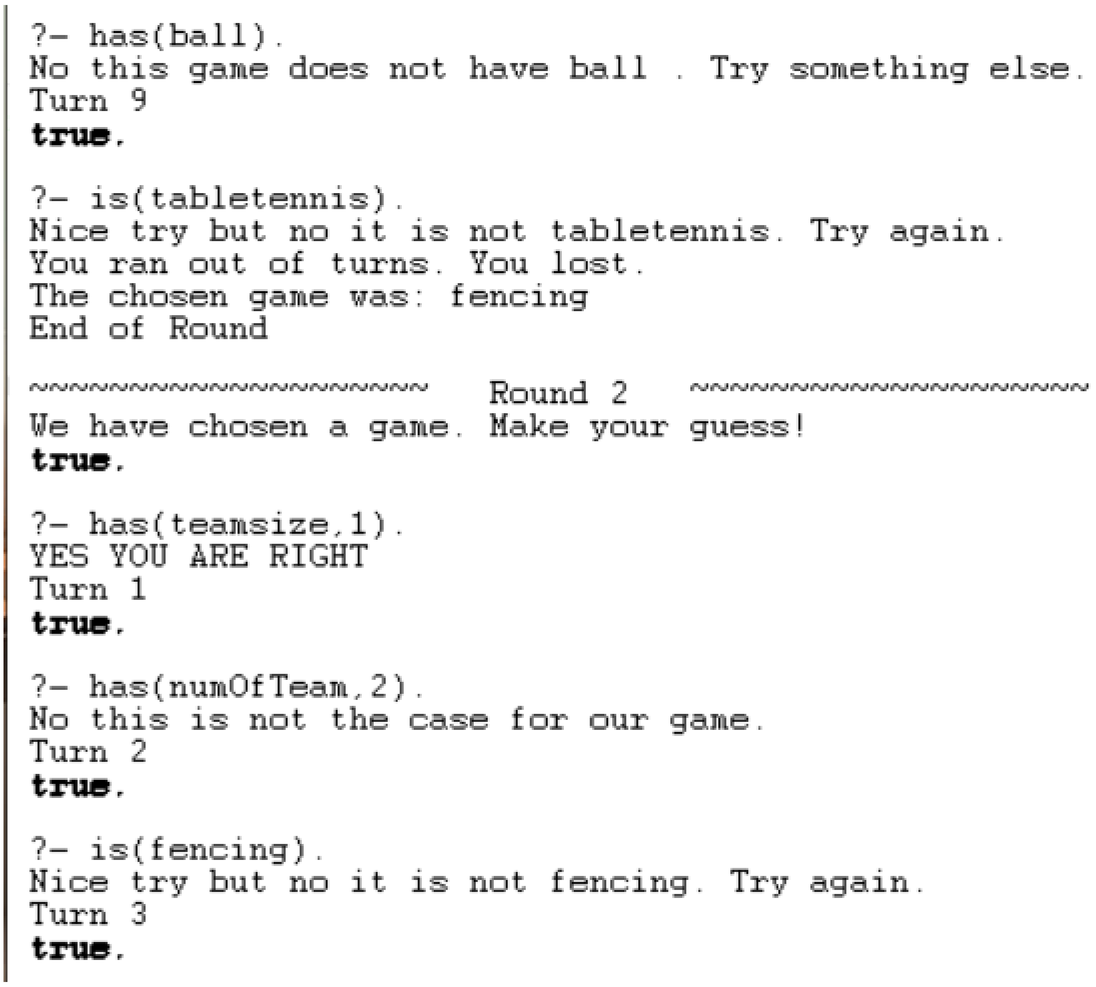
Cases demonstrated in the figures:

Figure	Showcasing points
1	• Showing intro screen • has/2, case 1 – correct key word and value —-> print correct message • has/2, case 2 – wrong key word -> support messages • has/2, case 3 – right keyword, wrong value -> print incorrect message • turn increment
2	• has/1 – fail case • is/1 – fail case • Failing a round by reaching N == 10 – print message and chosen game • Round increment • Choosing new game
3	• has/1 - success case • is/1 - success case • Winning a round as it reaches turn 10 (boundary case)
4	• Showing termination of case • Winning at turn < 10 case • End of Game alert • Restart command



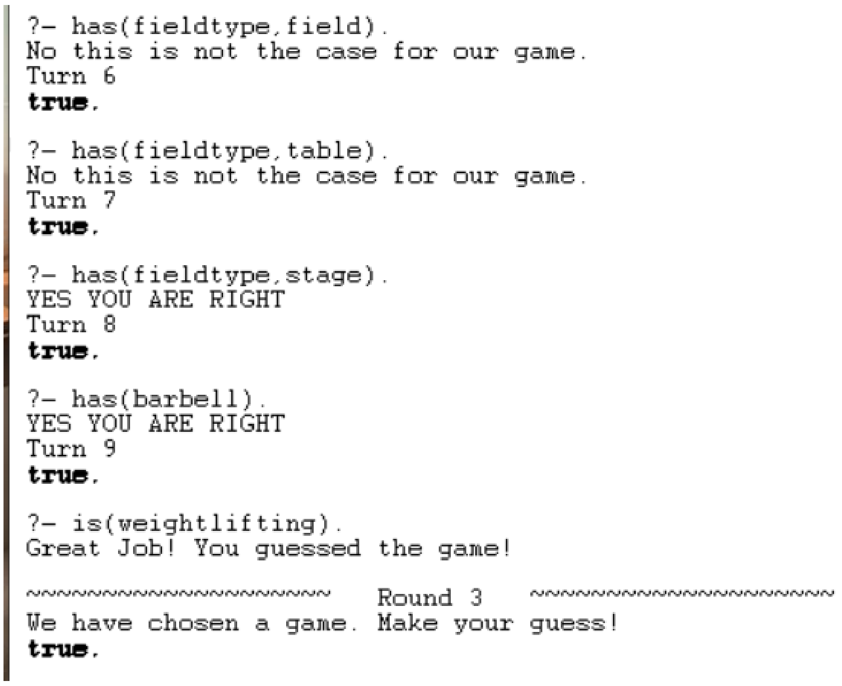
```
% KB_facts.pl compiled 0.00 sec, 18 clauses
Welcome to 10 Questions! - A game built on Prolog.


In this game, your role is a Questioner and the program acts as a Answerer.
The Answerer has come up with a Summer Olympic Game. Your job is to find out which game it is wit
hin 10 questions.
You are to use Prolog queries to ask the questions. The queries available:
        1. has(value), for e.g: has(ball), has(12), has(racquet), etc. We allow ambiguity so as t
o give you more room
        2. has(keyword,value), for e.g: has(equipment, ball), has(fieldtype,court). There is supp
orted keyword list you will find out if you use the wrong keyword. 'many' can be used as a value
if you need to.
        3. is(X), for e.g: is(Tennis). This is the only decision query to be made. It also counts
 as a turn.

Do note that we lowercase the game name - tabletennis, waterpolo. Some game has double version, b
ut for ease we don't include double as game here for simplicity. There are total of 17 games in t
he database in order to ensure each game is uniquely identifiable by its 5 attributes.
There will be 5 rounds in total.
All the best~

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


~~~~~~~~~~~~~~~~~~~~   Round 1   ~~~~~~~~~~~~~~~~~~~~
We have chosen a game. Make your guess!
% c:/Users/hoang/Documents/Prolog/Asg2/10Q_v9.pl compiled 0.00 sec, 15 clauses
?- has(teamsize,1).
YES YOU ARE RIGHT
Turn 1
true.

?- has(teamnum,2).
Supported keywords are:
[teamsize,numOfTeam,fieldtype,equipment,gameMode]. You are still counted as wrong though.
Turn 2
true.

?- has(numOfTeam,2).
YES YOU ARE RIGHT
Turn 3
true.

?- has(equipment,none).
No this is not the case for our game.
Turn 4
true.
```

Figure 1



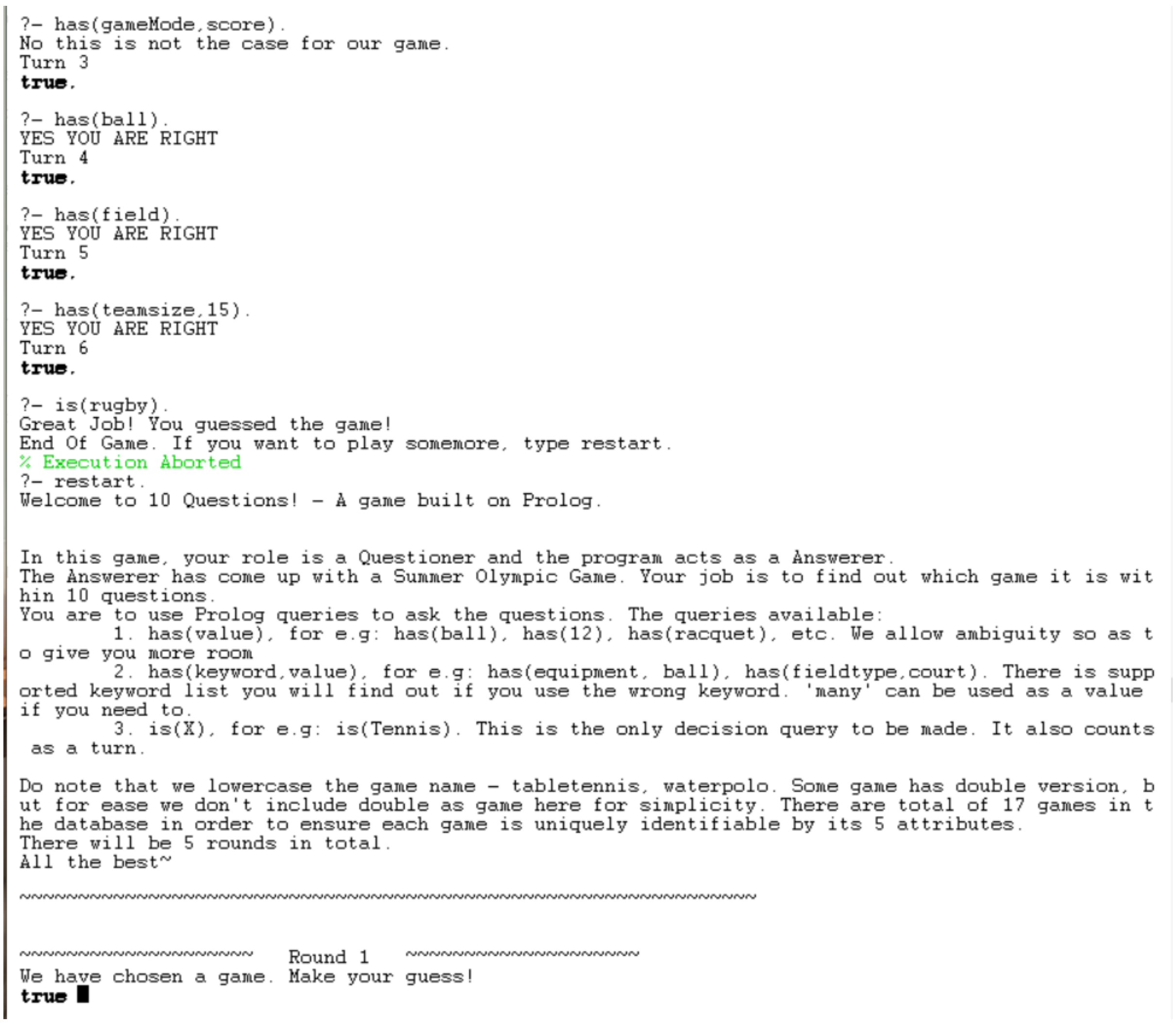
```
?- has(ball).
No this game does not have ball . Try something else.
Turn 9
true.

?- is(tabletennis).
Nice try but no it is not tabletennis. Try again.
You ran out of turns. You lost.
The chosen game was: fencing
End of Round

~~~~~~~~~~~~~~~~~~~~~~~~~~~   Round 2   ~~~~~~~~~~~~~~~~~~~~~~~~~~~
We have chosen a game. Make your guess!
true.

?- has(teamsize,1).
YES YOU ARE RIGHT
Turn 1
true.

?- has(numOfTeam,2).
No this is not the case for our game.
Turn 2
true.

?- is(fencing).
Nice try but no it is not fencing. Try again.
Turn 3
true.
```

Figure 2



```
?- has(fieldtype,field).
No this is not the case for our game.
Turn 6
true.

?- has(fieldtype,table).
No this is not the case for our game.
Turn 7
true.

?- has(fieldtype,stage).
YES YOU ARE RIGHT
Turn 8
true.

?- has(barbell).
YES YOU ARE RIGHT
Turn 9
true.

?- is(weightlifting).
Great Job! You guessed the game!

~~~~~~~~~~~~~~~~~~~~~   Round 3   ~~~~~~~~~~~~~~~~~~~~~
We have chosen a game. Make your guess!
true.
```

Figure 3



```
?- has(gameMode,score).
No this is not the case for our game.
Turn 3
true.

?- has(ball).
YES YOU ARE RIGHT
Turn 4
true.

?- has(field).
YES YOU ARE RIGHT
Turn 5
true.

?- has(teamsize,15).
YES YOU ARE RIGHT
Turn 6
true.

?- is(rugby).
Great Job! You guessed the game!
End Of Game. If you want to play somemore, type restart.
% Execution Aborted
?- restart.
Welcome to 10 Questions! - A game built on Prolog.


In this game, your role is a Questioner and the program acts as a Answerer.
The Answerer has come up with a Summer Olympic Game. Your job is to find out which game it is wit
hin 10 questions.
You are to use Prolog queries to ask the questions. The queries available:
        1. has(value), for e.g: has(ball), has(12), has(racquet), etc. We allow ambiguity so as t
o give you more room
        2. has(keyword,value), for e.g: has(equipment, ball), has(fieldtype,court). There is supp
orted keyword list you will find out if you use the wrong keyword. 'many' can be used as a value
if you need to.
        3. is(X), for e.g: is(Tennis). This is the only decision query to be made. It also counts
 as a turn.

Do note that we lowercase the game name - tabletennis, waterpolo. Some game has double version, b
ut for ease we don't include double as game here for simplicity. There are total of 17 games in t
he database in order to ensure each game is uniquely identifiable by its 5 attributes.
There will be 5 rounds in total.
All the best~

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


~~~~~~~~~~~~~~~~~~~~~~~~~     Round 1    ~~~~~~~~~~~~~~~~~~~~~~~~~
We have chosen a game. Make your guess!
true
```

Figure 4