Assignment 1: Ten questions

1 Introduction

In this laboratory session, a prolog program called “Ten questions” is built. In this program, the program will act as the answer by randomly choosing a sport. Once chosen a sport, the user will act as the questioner and query based on the properties of the sports. The prolog program will answer if it is true or false. A maximum of ten queries are allowed before the user must guess what the sport is the prolog program has chosen.

2 Facts in the program

There are several facts in the program. They vary from the sports available, properties of the sports, temporary data to be stored, and many more.

2.1 Static facts

Static facts are facts that will remain unchanged throughout the game. These facts are created based on the problem statements and requirements of the program. Below are the static facts in the program.

sport() —These are the sports that the program can randomly choose from. Example:

- sport(badminton).
- sport(basketball).

properties() —These contain properties of the sports that are available. User can query properties which will be used to compare with the properties of the chosen sport. Example:

- properties([racket, ball, net, court, singles, doubles, teams_per_game=2], tennis).
- properties([ball, jersey, cap, goalpost, pitch, teamsize=15, teams_per_game=2], rugby).

2.2 Dynamic facts

Dynamic facts are like variables. They are used to temporary store data which the content might varies through the running lifetime of the program. Example:

gameCounter() — The game counter keep tracks of the current round of the game. The game counter starts from one and keep increasing whenever the user chooses to play another round. The game counter will only be reset in when the user chooses not to play another round.

- :- dynamic(gameCounter/1).
- gameCounter(1).

triesLeft() — The tries left counter keep tracks of the number of tries the users left to query the program. Each time the user queries the program, the number of tries will decrease and display to the user.

- :- dynamic(triesLeft/1).
- triesLeft(10).

chosenSport() — The chosen sport will store the current sport that is randomly chosen by the program. This will be useful to keep track of the chosen sport for that round when user guess the sport or query the properties of the sport.

- :- dynamic(chosenSport/1).

3 Flow chart

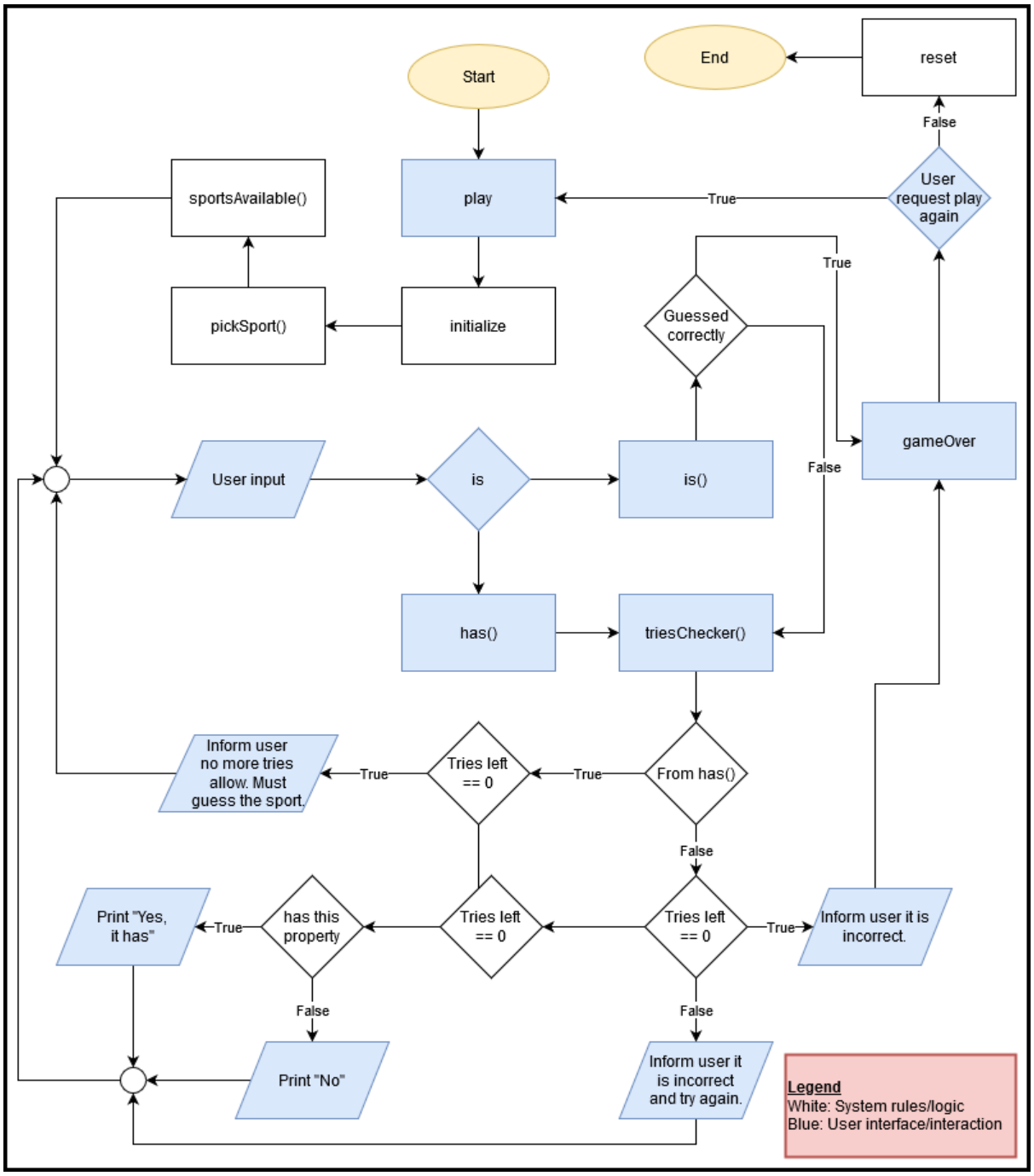

Figure 3: Flow chart of the prolog program

4 System logics

System logic is important as it sets the logic of how the program process data behind the scenes. Despite there are several system logics in the code, we will only be focusing on the main ones which are shown in the flow chart (see Figure 3).

4.1 initialize

initialize is always called when a new round begins. In *initialize*, a sport will be randomly picked from the list of available sports from *pickSports()*. This list of available sports are the sports that has not being chosen in the previous round. This ensures that there will not be repeated sports chosen.

In initialize, the previously selected sport will be retracted and the randomly picked sport will be asserted into *selectedSport()* as a fact. This will be useful during player interaction with the program as *selectedSport()* will act as the temporary variable to the currently selected sport for the round.

The *triesLeft()* is a dynamic variable which stores the number of tries left for user to query the program for that round. In initialize, the number of tries left in *triesLeft()* will reset back to 10 before the user begin to play that round.

```
39  /* initilize a new round such as pick a new sport,
40   temporary store the selected sport for the round,
41   and reset number of tries left for query for the user */
42  initialize :-
43      pickSport(SportPicked),
44      retractall(selectedSport(_)),
45      assertz(selectedSport(SportPicked)),
46      retractall(triesLeft(_)),
47      assertz(triesLeft(10)).
```

4.2 pickSport()

In *pickSport()* a sport will be randomly picked from the list of sports from *sportsAvailable()*. Once the sport has been picked, it will be asserted into *chosenSport()*. The chosenSport() is dynamically added whenever a new sport is picked allowing the program to keep track of the chosen sports in previous rounds. This prevents sports from being repicked in subsequent rounds.

```
74  /* randomly generate an available sport from a list of sports available to play.
75   The chosen sport will be keep tracked in chosenSport() */
76  pickSport(Result) :-
77      sportsAvailable(X),
78      random_member(Result, X),
79      assertz(chosenSport(Result)).
```

4.3 sportsAvailable()

In *sportsAvailable()*, two kinds of list of sports are generated. The first list generated is the list of all sports in the program. The second list generated is the list of sports that was picked by the program in previous rounds. The lists are subtracted to obtain the list of available sports. These available sports are sports that have not be chosen before.

```
96  /* find all available sports left since cannot have repeated chosen sports */
97  sportsAvailable(ValidSportsList) :-
98      findall(X, sport(X), SportsList),
99      findall(Y, chosenSport(Y), ChosenSportsList),
100     subtract(SportsList, ChosenSportsList, ValidSportsList).
```

4.4 reset

In reset, all newly added dynamic facts during the game are retracted from the knowledge base. The dynamic facts that were initially in the program are reset back to the original values.

Newly added facts:

- chosenSport()

Dynamic facts that existed since the beginning:

- gameCounter()
- triesLeft()

```
81  /* reset knowledge base */
82  reset :-
83      retractall(chosenSport(_)),
84      retractall(gameCounter(_)),
85      assertz(gameCounter(1)),
86      retractall(triesLeft(_)),
87      assertz(triesLeft(10)),
88      abort.
```

5 User interfaces

The user interfaces are prolog rules that have direct interaction with the user. These prolog rules may be rules that the user can directly call from the prolog console. These rules may also be rules that print output to user.

Prolog rules user can directly call:

- play
- has()
- is()

Prolog rules that prints output to user only:

- triesChecker()
- gameOver

5.1 play

play is the starting interface for the user to use the program. *play* is needed to start a game or begin a new round. *play* will calls initialize and print to the user the round of the game. *play* will then calls *increment* which will increment the *gameCounter()* to keep track of the round of the game.

```
6   /* command to start the game or new round */
7   play :-
8       initialize,
9       gameCounter(X),
10      format("Begin round ~w.", X),
11      increment.
```

5.2 has()

has() can be called by the user to query the program. When querying the program, user can query the property of the sports. These including querying if the sport has a pool. To query a pool: **has(pool)**. The queried property will be used to check if it is a member of the properties list of the chosen sport If the property exists for the chosen sport, the program will print to the user "**Yes, it has.**" along with the number of tries left. Else, "**No.**" will be printed along with the number of tries left. The number of tries left is the number of tries the user left to query the program. At the start of *has()*, the number of tries will decrement by calling *decrement*.

```
13  /* will decrease tries left before checking if guessed property is correct or not */
14  has(GuessProperty) :-
15      triesChecker("has()");
16
17      decrement,
18      selectedSport(SportPicked),
19      properties(PropertyList, SportPicked),
20      member(GuessProperty, PropertyList),
21      triesLeft(X), format("Yes, it has. (Left ~w tries.)", X);
22
23      triesLeft(X), format("No. (Left ~w tries.)", X).
```

5.3 is()

is() can be called by the user to guess the sport randomly chosen by the program. Example, user can guess if the sport is swimming by inputting this in the console: **is(swimming)**. If the sport guess by the user is correct, the program will output a congratulation message and calls *gameOver*. Else the program will notify the user that the guess is incorrect. *is()* will also call *triesChecker()* which will check the number of tries left.

```
25  /* will decrease tries left before checking if guessed sport is correct or not.
26   check if still have tries left before deciding if continue or game over */
27  is(GuessSport) :-
28      selectedSport(GuessSport),
29      write("Congratulations! You have correctly guessed the sport."), nl,
30      gameOver;
31
32      triesChecker("is()");
33
34      write("Incorrect. Please try again."), nl.
```

5.4 triesChecker()

The *triesChecker()* checks the number of tries left for the user to query the program. Once it reaches zero tries left, the user will have to guess what the sport is. If the guess is incorrect, the round will end where the program will call *gameOver* to displays “Game over!”. If the *triesChecker()* is called from *has()*, it means the user tried to query the program. If there are no more tries, the *triesChecker()* will prompt the user to guess the sport. If the *triesChecker()* is called from *is()* and it has zero tries left, it means the user made the final attempt to guess the sport. The round will terminate.

```
/* check if user has run of tries */
triesChecker(From) :-
    triesLeft(X),
    From == "has()",
    X == 0 -> write("No more questions available. Please guess what sport it is."), nl;

    triesLeft(X),
    From == "is()",
    X == 0 -> write("Sorry! It's incorrect."), nl,
    gameOver.
```

5.5 gameOver

In *gameOver*, the list of available sports left are obtained from *sportsAvailable()*. Based on the obtained list, the list is checked if it is empty. If the list is empty, it means all the sports have been chosen and there are no more games left. User has successfully cleared all the available rounds. “**Game over!**” will be printed to the user and reset the knowledge base by calling *reset*. If the list is not empty, the program will prompt if the user would like to play another round. If the user input “y”, *play* will be called to start another round in *restartGame()*. If the user input “n”, *reset* will be called to reset the knowledge base.

```
/* end of the round, check if still have sports left to guess to play more rounds */
gameOver :-
    sportsAvailable(ValidSportsList),
    ValidSportsList \== [],
    write("Game over! Would you like to restart the game? (y/n) "),
    read(Input),
    restartGame(Input);

    write("Game over!"), nl,
    reset.
```

6 Demonstration

```
?- play.
Begin round 1.
true.

?- has(ball).
Yes, it has. (Left 9 tries.)
true .

?- has(court).
No. (Left 8 tries.)
true.

?- has(field).
Yes, it has. (Left 7 tries.)
true .

?- has(goalpost).
Yes, it has. (Left 6 tries.)
true .

?- has(cap).
No. (Left 5 tries.)
true.

?- is(football).
Congratulations! You have correctly guessed the sport.
Game over! Would you like to restart the game? (y/n) y.
Begin round 2.
true .

?- has(field).
No. (Left 9 tries.)
true.

?- has(court).
Yes, it has. (Left 8 tries.)
true .

?- has(basket).
Yes, it has. (Left 7 tries.)
true .

?- is(basketball).
Congratulations! You have correctly guessed the sport.
Game over! Would you like to restart the game? (y/n) y
|: .
Begin round 3.
true .
```

```
?- has(pitch).
Yes, it has. (Left 9 tries.)
true .

?- has(stick).
No. (Left 8 tries.)
true.

?- has(ball).
Yes, it has. (Left 7 tries.)
true .

?- is(rugby).
Congratulations! You have correctly guessed the sport.
Game over! Would you like to restart the game? (y/n) y
|: .
Begin round 4.
true .

?- has(cap).
No. (Left 9 tries.)
true.

?- has(court).
Yes, it has. (Left 8 tries.)
true .

?- has(racket).
Yes, it has. (Left 7 tries.)
true .

?- has(net).
Yes, it has. (Left 6 tries.)
true .

?- is(tennis).
Congratulations! You have correctly guessed the sport.
Game over! Would you like to restart the game? (y/n) y.
Begin round 5.
true .

?- has(field).
Yes, it has. (Left 9 tries.)
true .

?- has(bat).
Yes, it has. (Left 8 tries.)
true .


?- is(baseball).
Congratulations! You have correctly guessed the sport.
Game over! Would you like to restart the game? (y/n) n
|: .

% Execution Aborted
?-
```

Conclusion

In conclusion, the prolog program can be improved by creating Graphical User Interface (GUI) to make the program more user friendly. It was not created due to time constraint. The program can also be improved by break downing the number properties of sports into separated facts so in future new properties of existing sports can be easily asserted into the knowledge base.