Laboratory: Programming Multi-Agent Systems with ASTRA

The second lab on ASTRA programming is focused on agent communication using FIPA-ACL. That is, in this exercise we will develop a multi-agent system (system with more than one agent) in which the agents communicate with one another via an agent communication language.

The specific problem we will tackle in this lab is to develop agents that can play the game Tic-Tac-Toe. In this lab, we will focus mainly on the representation of the game (remember the domain modelling activity) and on implementing a custom protocol to models the in-game interaction between the agents.

As in the previous lab, the agents will maintain a mental model of the game board rather than using an explicit shared resource. Also, while there are many valid ways of implementing the agent programs in this lab, I would ask you to again try and base your program on the **practical reasoning** model we covered in previous labs.

Marking Scheme

Complete 1 part	D
Complete 2 parts	C
Complete 3 parts	B
Complete ALL parts	A

Submission Instructions

Create a new maven project for each part. The name of the project folder should be Part<num>. For example, the answer to Part 1 should be in a folder Part1. Each part folder should be in a folder called Lab2. At the end, please ZIP the Lab2 folder up and submit via moodle.

The Problem: Tic-Tac-Toe

Tic-Tac-Toe is a long established problem in Artificial Intelligence, the value of which is highlighted in Wargames (1983) where it is used to stop Global Thermonuclear War.

You can watch the relevant clip here: https://www.youtube.com/watch?v=NHWjlCaIrQo

While we don't have these lofty goals, Tic-Tac-Toe proves to be an elegant and simple problem with which to explore agent communication. The reason: Tic-Tac-Toe is a two-player game. To implement it, we need two agents…

Tic-Tac-Toe was used as an example in both the Domain Modelling activity and the Lesson on Programming Agents in AgentSpeak(L). In fact, a complete AgentSpeak(L) program was developed as part of the second lesson and can be seen on slide 36 (of 37). If you have not watched this lecture, it is essential that you do so before attempting this lab.

Part 1: Translation to ASTRA

The first task to complete is to translate the program from AgentSpeak(L) to ASTRA. The code should be placed in a file called `Player.astra`. Once you have done this, use the `initial` statement to specify a number of test board configurations and write a main rule that prints out one of the following:

- Game won by player <X>
- Game lost by player <X>
- Game is drawn
- It is <X>'s turn
- Game has not started

There are two main approaches to implementing the main rule: overloading the rule with a different context for each scenario above; or using a single rule together with the if statement.

There are two small complications in transforming the code from AgentSpeak(L) to ASTRA:

(a) Currently, we have no environment, so there is no move(T, L) action (this is used in the +turn(X) rule). For the purposes of this program, create another rule to handle the goal: `!move(string T, int L)`. This rule should update the game state (add the `played(…)` belief and drop the `turn(…)` belief (because it is now the opponents turn). For testing purposes, you can also print out something to indicate that the move was made via a `console. println(…)` statement (e.g. token X played at location Y). In the configurations where it is the players turn, this should result in the statement being executed and the message being displayed on the dashboard.

(b) In the example code, I promote the use of an inference rule to model empty locations. This does not work – you should enumerate all the free locations (one belief per location). This will obviously have an impact on (a) because once a token is played, the location is no longer free.

Comment out all but one of the configurations (which one does not matter).

NOTE: Remember to set the astra.main property to Player so that your code is executed via astra:deploy.

Part 2: Creating a Multi-Agent System

The second task is to create a multi-agent system consisting of 3 agents: the main agent (which we run to start the program), and 2 Player agents who will play Tic-Tac-Toe against one another.

Creating agents in ASTRA is done via the System API (module). To create an agent simply write:

```
system.createAgent("name", "ASTRAClass");
```

You can then give the agent you have created a `!main(…)` goal with by writing:

```
system.setMainGoal("name", [list, of, arguments]);
```

To complete this task, create a `Main.astra` program that includes a main rule (remember to modify your pom.xml file to . In that rule use the above statements to create two agents: player1 and player2. Initialise player1 to be the O token and for it to be their turn. Initialize player2 to be the X token.

HINT: The arguments can be heterogeneous (e.g. ["O", true] vs ["X", false]) and you can match a specific pattern of arguments in the main (e.g. rule +!main([string token, boolean turn]).

Modify the Player code to set up an initial game state (no tokens on the board) with the token and turn beliefs being based on the arguments provided in the main goal (its okay if the player whose turn it is makes a move).

Part 3: The Game Protocol

The third task is to implement the protocol that will be used to implement turn taking in the game. For now, we will use a FIPA Inform Message. Once an agent decides what move they will make (via the turn(X) rule), the agent should update their internal model (via the !move(T, L) goal) and then send an `inform` message to its opponent with the content `played(T, L)`.

To do this, you need to add the name of the name of the opponent to the parameters passed to the main rule. You will also need to add an `opponent(string)` belief to store the name of the opponent agent so that you can retrieve it when you need to send the `inform` message.

Another rule must be implemented to handle the receipt of the message. This message rule, with event `@message(inform, string sender, played(string T, string L))` should update the game state, and adopt the turn belief so that the player can make their next move.

Once there are no more free squares, the game should stop and each agent should print out the outcome of the game (winner/loser or drawn).

Part 4: Jazzing it up

The last task is to jazz up the game a little by introducing players with different strategies. To achieve this, you need to use the inheritance mechanism of ASTRA. Start by creating a new program called `LinearPlayer.astra`. This program should look as follows:

```
agent LinearPlayer extends Player {

}
```

Copy all the strategy rules (the `+!move()` rules) into the new program and delete them from `Player.astra`. Modify the Main agent to use `LinearPlayer` instead of `Player`. The game outcome should be the same.

Once you have this working, create another program called `Opponent.astra` and implement an alternative strategy that can beat the strategy specified in `LinearPlayer.astra`. Modify you Main rule to may player2 be the `Opponent` agent.

Finish by trying to create a better strategy in a file called Better.astra. Describe your strategy in a set of comments added to the file.