Laboratory: Programming with Environments

This lab about how to program ASTRA agents that are situated in an environment. That is, in this exercise we will develop a single agent system that will control some entities residing in an environment in order to achieve some well-defined task(s).

The specific problem we will tackle in this lab is to develop an agent that can control a gripper which it can use to build block towers (again think back to the domain modelling problems we discussed). To support this, the agent can perform two basic actions:

- `pickup(X)`: the gripper is used to pickup block X; assuming it is not already holding X.
- `putdown(X, Y)`: the gripper is used to put block X on top of Y; assuming it is already holding X and Y is free (or the table).

Some further details of the environment can be found here:
https://github.com/eishub/tower/blob/master/doc/TowerEnvironment.pdf

This lab shifts the focus from the maintenance of mental models to the use of explicit resources. As with the previous lab, I ask to you to again to use the **practical reasoning** style of programming covered in previous labs.

Marking Scheme

Complete 1 part	E
Complete 2 parts	D
Complete 3 parts	C
Complete 4 parts	B
Complete ALL parts	A

-/+ based on quality of solution/use of practical reasoning from part 4 onwards.

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 Brightspace.

Part 1: Connecting up the Environment

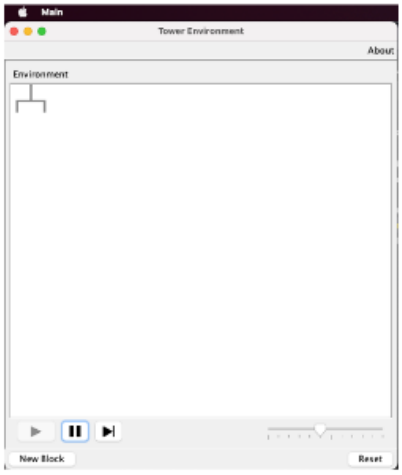
The first task is get yourself set up with the Tower World environment. The best way to do this is to download the example project from Brightspace.

Once you have a copy of the project on your computer, run it by typing “mvn”.

Notes:

- *Don’t add any goals to the mvn command – the project already comes with a default goal that includes a number of steps required for the code to run correctly.*
- *Make sure you are using **Java 1.8**. If not, you will get a classloader exception when you run the code.*

When the program runs correctly, you should see an interface like the one of the left below, and the console output should look something like the output on the right.




```
s/EIS/astra-eis-tower/target/dependency/astra-interpreter-1.1.1.jar
[INFO] Copying jackson-core-2.12.4.jar to /Users/remcollier/Git/astra-language/examples/EI
S/astra-eis-tower/target/dependency/jackson-core-2.12.4.jar
[INFO] Copying jackson-annotations-2.12.4.jar to /Users/remcollier/Git/astra-language/exam
ples/EIS/astra-eis-tower/target/dependency/jackson-annotations-2.12.4.jar
[INFO] Copying jackson-databind-2.12.4.jar to /Users/remcollier/Git/astra-language/example
s/EIS/astra-eis-tower/target/dependency/jackson-databind-2.12.4.jar
[INFO] Copying junit-4.12.jar to /Users/remcollier/Git/astra-language/examples/EIS/astra-e
is-tower/target/dependency/junit-4.12.jar
[INFO] Copying hamcrest-core-1.3.jar to /Users/remcollier/Git/astra-language/examples/EIS/
astra-eis-tower/target/dependency/hamcrest-core-1.3.jar
[INFO] Copying astra-apis-1.1.1.jar to /Users/remcollier/Git/astra-language/examples/EIS/a
stra-eis-tower/target/dependency/astra-apis-1.1.1.jar
[INFO] Copying astra-compiler-1.1.1.jar to /Users/remcollier/Git/astra-language/examples/E
IS/astra-eis-tower/target/dependency/astra-compiler-1.1.1.jar
[INFO] Copying astra-eis-1.1.1.jar to /Users/remcollier/Git/astra-language/examples/EIS/as
tra-eis-tower/target/dependency/astra-eis-1.1.1.jar
[INFO] Copying eis-0.5.0.jar to /Users/remcollier/Git/astra-language/examples/EIS/astra-ei
s-tower/target/dependency/eis-0.5.0.jar
[INFO] Copying tower-1.3.0.jar to /Users/remcollier/Git/astra-language/examples/EIS/astra-
eis-tower/target/dependency/tower-1.3.0.jar
[INFO]
[INFO] --- astra-maven-plugin:1.1.1:deploy (default-cli) @ astra-eis-tower ---
[Scheduler] Processors: 10
[main]Press New Block until block d is created...
```

The final step in this part is to press the New Block button repeatedly until block 'D' is created. When this happens, a "You did it" statement will be printed to the console. You should also see 4 blocks in the interface.

Part 2: Working with Blocks

The second task aims to get you interacting with the environment. We will explore this by adding some statements to the end of the `+!main(…)` rule provided in the code for Part 1. Start by copying your answer to Part 1 into a folder called "Part2".

1. **Picking up a block:** To pick up block A, for example, you should use the following statement:

   ```
   ei.pickup("a");
   ```

   Append this code to the end of the `+!main(…)` rule and run the program. What happens?

   Notice that all the statements are prefixed by "ei.". This matches the name of the EIS module we created at the top of the program. You should read these statements as: "Execute the action specified after the period in the EIS environment that this module is connected to". As a warning, the EIS module has some pre-defined statements(e.g. launch(…), join(), link(…), startEnv()). The action parts of these statements should be treated as reserved actions and should not be associated with environment actions. In the rare cases where a reserved action is used, you can use the ei.perform(…) reserved statement (but you should not need this for this lab).

2. **Putting a block down:** To put a block you are holding on top of something, you should use the `ei.putdown(…)` statement. For example, to put the block you picked up in (1) down you should add the following statement to the end of the `+!main(…)` rule:

   ```
   ei.putdown("a", "b");
   ```

   Run the program and see what happens…

This program does not work because actions in this environment are durative (they take some time) and non-blocking (the statement triggers the action, but does not block until the action is completed).

Not all EIS environments work this way, but when they do, we need to make the agent block its intention until the action is completed. In ASTRA, we can achieve this by using a `wait(…)` *statement.*

To get this program to work, we need the agent to wait until the first `ei.pickup(…)` action is finished. We know it has finished when it is holding the block we attempted to pick up. We can implement this by adding the following statement immediately after the `ei.pickup("A")` statement:

```
wait(ei.holding("a"))
```

Again note the "ei." prefix. This should be read as "Evaluate the belief specified after the period against the current set of perceptions associated with the EIS environment that this module is connected to". This will be evaluated to true of false depending on whether or not a corresponding percept exists.

As a final step in (2), add a `println(…)` statement to the end of the program that prints out "FINISHED". Run the program to see what happens. You should see that "FINISHED" is output before the `ei.putdown(…)` action completes. This is because this action has been implemented in that same way as the `ei.pickup(…)` action. As a result, you also need to add a second wait(…) statement to force the agent to wait until the block has actually been put down:

```
wait(ei.on("a", "b"))
```

3. **Putting on the table:** To put a block on the table, you can use something like this:

```
ei.putdown("a", "table")
```

Modify your answer to make the agent print out "TOWER BUILT" after it puts A on B; sleep for 2 seconds; pick up A and put it on the TABLE; and finally print out finish.

4. **Handling Events:** The ASTRA EIS implementation keeps the perceptions it receives from the environment separate to the internal beliefs of the agent (this is why we have to use the `ei` prexfix when querying the state of the environment). The EIS module implements custom events that are the EIS equivalent of the belief adoption/retraction events. The format of the event looks like this:

```
+(-)$ei.event(funct belief)
```

Extend your program to include the following rule and run the program to see what happens:

```
rule +$ei.event(funct belief) {
    C.println("New Perception Event: " + belief);
}
```

Now change the + to a – and run it again.

As you can see, the first version of the rule relates to events that the agent has a new perception about the state of the environment. The second version of the rule refers to removal of a

perception. The above rule captures ALL events from the environment, to capture a specific event, you can do something like this:

```
rule +$ei.event(on(string A, string B)) {
    C.println(A + " has been placed on: " + B);
}
```

We could also modify the original program as follows:

```
rule +!main(list args) {
    ei.launch("hw","dependency/tower-1.3.0.jar");
    ei.init();
    ei.start();
    ei.link("gripper");

    C.println("Press New Block until block d is created...");
}

rule +$ei.event(block("d")) {
    C.println("You did it!");
}
```

Notice – we respond to the event of block D being added rather than waiting for the perception to be adopted. Modify your answer above to use this event rather than the `wait(…)` statement.

NOTE: Pay attention to rule order as the more specific rules have to come before the more general rules in the code (otherwise the code won't work the way you expect it to)

Part 3: 3-block Towers

1. **Building a 3 block tower:** Now that we have a basic understanding of the problem domain, it is time to start programming goals into the agent. The first goal to program involves getting the agent to build a 3 block tower, where block C is on the table, block B is on block C and block A is on block B.

   As a first step, try to build this tower by modifying your answer to part 2. Run the code to see if the correct tower is created.

2. **Generalising tower construction:** We can generalise the construction of a three block tower by specifying a goal

   ```
   !tower(string A, string B, string C)
   ```

   This goal specifies some general tower where C is on the table; B is on C and A is on B. Write a rule to handle the event +!tower(string A, string B, string C) based on your answer to (1). Modify the rule handling the event `block("d")` to create the same tower using:

   ```
   !tower("a", "b", "c")
   ```

   Run the program to check that it produces the same output.

3. **Multiple Towers:** Add a second `!tower(…)` goal after the first one: `!tower("d","c","b")`. Does it build 2 towers? Why not? *Comment this additional goal out after you try it.*

Part 4: Getting all Practical Reasoning about it

Up to now, we have focused on the mechanics of how to build towers, but we have not thought about how we would do it with ***practical reasoning***. As a reminder, practical reasoning is about using goals to specify future states of the environment that the agent should attempt to achieve.

Here, future states are linked to the environment rather than the internal state of the agent. This can be a little confusing because the perceptions only provide basic knowledge about the environment (holding, on, …) and some knowledge is derived from that basic knowledge. The specification derived knowledge can be realised through inference rules.

For example, the concept of a block being free (e.g. `free("a")`) does not exist in the tower environment, but can be derived from existing EIS perceptions:

```
inference free(string A) :- ~ei.on(string B, A)
```

Informally, this states that we can infer that block A is free if there is no perception of a block B that is on A.

Similarly, we can introduce the concept of a three block tower by using this inference rule:

```
inference tower(string A, string B, string C) :-
     ei.on(A, B) & ei.on(B, C) & ei.on(C, "table");
```

It is this second inference rule that is the starting point for building an agent that constructs towers using practical reasoning. Specifically, it tells us that the goal !tower(…) can be achieved if the three beliefs on the right hand side are true. This means that we can write the following rule:

```
rule +!tower(string A, string B, string C) {
    !on(C, "table");
    !on(B, C);
    !on(A, B);
}
```

Notice that the ordering of the on goals is inverted because when building a tower, order matters, but when reasoning about whether a tower exists, order does not matter!

Adopting a practical reasoning perspective. The next challenge is how to achieve the goal `!on(string A, string B)`. There are a number of scenarios – the easiest of which is that A is already on B:

```
rule +!on(string A, string B) : ei.on(A, B) {}
```

What about if A is not on B already? Well, the ideal situation is that the gripper is holding A and B is free, in which case you can simply perform the `putdown(…)` action:

```
rule +!on(string A, string B) : ei.holding(A) & free(B) {
    ei.putdown(A, B);
    wait(ei.on(A, B));
}
```

This is very neat because the conclusion of the plan is that the corresponding belief/perception become true (as is the idea for practical reasoning).

These rules describe the positive cases. A problem arises if the context is not true. This can be because (a) the agent is not holding block A or (b) because block B is not free. We handle these scenarios through two additional rules:

```
rule +!on(string A, string B) : ~free(B) {
    !free(B);
    !on(A, B);
}

rule +!on(string A, string B) : ~ei.holding(A) {
    !holding(A);
    !on(A, B);
}
```

Notice, each rule deals with one of the formulae of the context being false. The rule basically states that if formula X of the context is false, adopt a goal to achieve X and then re adopt the main goal. Each rule inserts a subgoal to cause the agent to act in a way that brings about the desired goal.

For example, an agent has the goal `!on("a", "b")` but is not holding "a", so it refines the plan to: `!holding("a");!on("a", "b")`. If block "b" was also not free, it would refine the plan as follows: `!free("b");!on("a", "b")`. If block "b" was not free AND the agent was not holding "a", it would start by freeing "b" and then picking up "a".

Notice that if I swapped the order of the above rules, then in this latter scenario, I would have to: pickup "a", put "a" down (so I can free "b"), and then pickup "a" again; so rule order even matters here!

In defining the rule that will achieve the goal `!on(A, B)`, you should try to ensure that the context specifies only the circumstances in which the rule will be successful (remember is a guard to constrain the use of the rule). This context can also be used in a practical reasoning sense to determine what additional means-end reasoning rules are required.

NOTE: In the previous labs, means-end rules typically manipulated some mental model held within the beliefs of the agent. In this lab, means-end rules should invoke actions to achieve the required state change via manipulation of the environment.

Following on from this, you will need to develop rules to handle the goals `!holding(string A)` and `!free(A)`. In both cases, try to start by writing the positive cases and the focus on the cases where some kind of refinement step is required.

The final program should work in the same way as part (3), however it should now be able create both towers (Part3 – step 3). Check that it does.

Part 5: Everythings Blockilicious!

The final task is to revise your program to work with towers of any size (so long as the blocks exist). Start by modifying the `!tower(…)` goal to reflect a tower of unknown height (you will need to use **lists** for this – see the practical reasoning slides).

Next, modify the deliberation rules so that they trigger the creation of towers of different heights (you can choose the specific tower configurations you use).

Finally, when the g block is created, modify the agent to satisfy the goal `!random_tower(…)` which creates and builds a random tower containing blocks a-g. Once the tower is built the agent should sleep for 2 seconds and then repeat (create and build another random tower, sleep and repeat again).