CHALMERS UNIVERSITY OF TECHNOLOGY

Artificial Intelligence project report

Group 20

Pierre BOUTRY, Dan DOLONIUS, Julien MICHELET, Emeric ROVERC'H

CHALMERS

1. Project Presentation

The goal of this project was to implement a dialogue system for controlling a robot living in a virtual block world. The robot we implemented can move around objects of different forms, colours and sizes and it can answer questions about the world and ask for clarifications whenever it finds the request ambiguous.

This has been performed using GrammaticaFramework (gf) for the grammar and java/prolog for the implementation of the application. The world is described using a JSON file and the executions of graphic animations are performed using Javascript.

Below is a macro explanation of how works the global application.

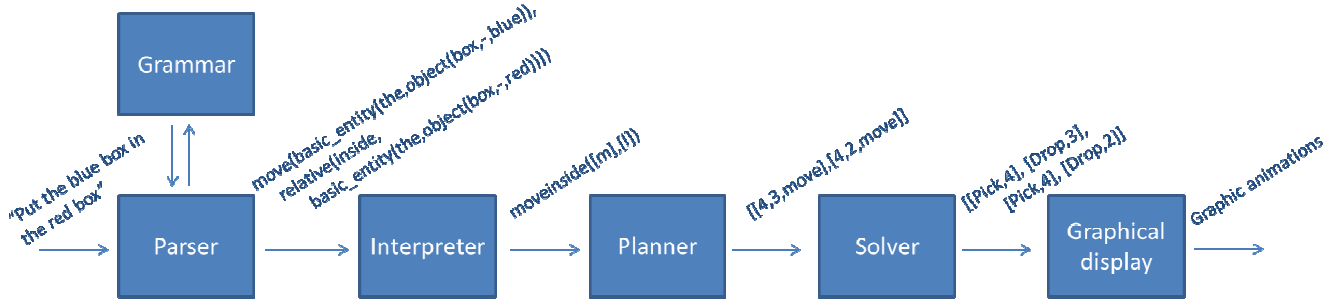
On client side:

> The user types a query as an input, which is handled with an ajax call in jquery. It is this particular ajax call that will execute our main function in prolog.

On sever side:

> The ajax call contains several attributes: world, objects, holding, state, utterance.
>
> All those information are stored in a JSON file that contains the world with the objects and the request. This JSON file is then read and handled using prolog.
>
> The first step is to transform the user input into a goal that is understandable by the interpreter. This is the role of the parser.
>
> Then the parsed query is sent to the interpreter which translates it to query understandable by the planner we implemented.
>
> Once the goal has been defined, we need to find a resolution plan. All the functions we need to perform movements have been implemented in the planner. This will give us a list of all the actions we need to perform to go from the original world to the requested one.
>
> Then this list of actions is handled by the solver which is going to produce a list of all the instructions required for the graphics rendering. The output is then sent back on client side for the graphics animations of all the steps.

The schema below gives a macro view of the steps of the applications:



2. The World

The world is represented by a floor on which several objects (of different forms, colors and sizes) can lay. The objects can stand in/on each other (if it is permitted by the world's laws).

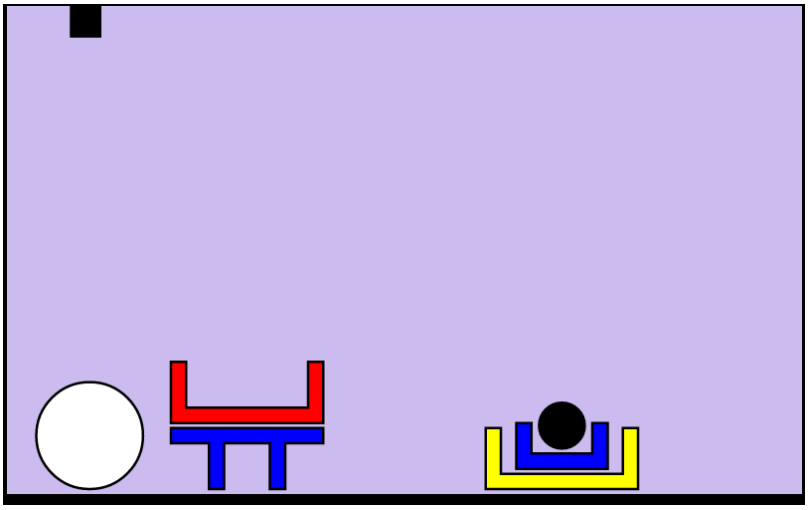
The goal of this project is to move around the objects, according to the request of the user, through a robot arm which can pick up and put down objects.

The floor is divided in n spaces, meaning that there is room on the floor for no more than n objects at the same time. Each of this space is represented as a column, so that the world can be described as a list of n columns of objects stacked on each other.

The world we implemented can contain all the objects of different forms, colours and sizes listed on the course homepage, plus the possibility to define an object of medium size (in order to create a more complex/realistic world).

Bellow the list of forms, colours and sizes available in "our" world:

- Forms: bricks, planks, balls, pyramids, boxes, tables.
- Colours: red, black, blue, green, yellow, white.
- Sizes: large, medium, small.



A JSON file describes the world as a list of columns of objects.

3. Improvements of the grammar

We started with the initial grammar (given at the beginning of the project) that we have completed and improved.

The major improvements we implemented are the possibility for the user to ask 3 new types of questions: where, what and count.

3.1. Where

The “where” question allows the robot to answer questions such as “where is the big white ball?” Or “where are the boxes?”. The synonyms for the “where” question are the words:Find, Where is and Where are.

3.2. What

With this “what” question, the robot can answer questions such as “what is under the red box?” Or “what are the object in the world?”. The synonyms for the “what” question are the words:What is and What are.

3.3. Count

This last question we added aims at permitting the user to get answers for requests such as “count the boxes in the world.” or “how many ball are in the world?”. The synonyms for the “count” question are:Count and How many.

3.4. Other improvements

Another major improvement we made from the original grammar is the possibility for the user to request actions and/or ask questions about stack instead of simply the whole world. The user can then perform requests of the form “what are the objects in stack 2?”, “count the small blue balls in stack 0.”,or “what are the objects on the right of stack 2?”

We also added 2 minor improvements to the original grammar: the alias “world” for “all the stacks” and the possibility for the user to put question marks at the end of his questions without having the robot answering with an error message.

Of course all these modifications regarding the grammar had to be echoed to the other layers of the application (parser and planner).

4. Interpreter

After the parser (and grammar) layer we then proceed to the interpreter layer.

The interpreter aims at translating the output of the parser into something understandable by the planner. For that it uses a tree to evaluate user query such as:
move(basic_entity(the,object(ball,-,black)),relative(inside,basic_entity(the,object(box,-,red))))

We make a set of rules to recursively satisfy every node in the tree, e.g:

```
Tree = a(b(c(d))).
interpret (a(X),Y)      :- interpret (X,Y), something1(Y).
interpret (b(X),Y)      :- interpret (X,Y), something2(Y).
interpret (c(X),Y)      :- interpret (X,Y), something3(Y).
interpret (d,Y)         :- something4(Y).
```

where Y is the variable which satisfies every node of the tree.

4.1. Interpretation rules

More explicitly we have implemented 62 rules, as below:

```
0  interpret(object(Type,Size,Color), World, @(null), Objects, SelectedObject) :-
1  interpret(object(Type,Size,Color), World, Holding, Objects, SelectedObject) :-
2  interpret(basic_entity(any,X), World, Holding, Objects, any(SelectedObject)) :-
3  interpret(basic_entity(the,X), World, Holding, Objects, [SelectedObject]) :-
4  interpret(basic_entity(all,X), World, Holding, Objects, SelectedObject) :-
5  interpret(relative_entity(any,X, Relation), World, Holding, Objects, any(SelectedObject)) :-
6  interpret(relative_entity(all,X, Relation), World, Holding, Objects, SelectedObject) :-
7  interpret(relative_entity(the,X, Relation), World, Holding, Objects, [SelectedObject]) :-
8  interpret(relative(beside,X), World, Holding, Objects, SelectedObject) :-
9  interpret(relative(leftof,X), World, Holding, Objects, SelectedObject) :-
[…]
15 interpret(relative(inside,X), World, Holding, Objects, SelectedObject) :-
16 interpret(absolute(beside,basic_stack(N)), World, Holding, Objects, SelectedObject) :-
[…]
21 interpret(absolute(inside,world), World, _Holding, _Objects, SelectedObject) :-
22 interpret(floor, _World, _Holding, _Objects, floor).
23 interpret(take(X), World, Holding, Objects,  take(SelectedObject)) :-
[…]
61 interpret(what(absolute(ontop, basic_stack(N))), World, Holding, Objects, whatontopstack([N])).
```

Where those rules can be understood as:

- 0-1 - Get object satisfying description.
- 2-4 - Handle the any or all cases for basic entities.
- 5-7 - Same as 2-4 but for relations as well.
- 8-15 - Get object which satisfies relation to "object X".
- 16-2 - Get object which satisfies relation to stack.
- 22 - floor is floor.
- 23-61 - Process output to goal.

Further we have some rules to see if an object is e.g. besides any other object:

```
isbeside(X,Y,World) :-
    member(ColS,World),member(X,ColS), nth0(IdxS,World,ColS),
    member(ColR,World),member(Y,ColR), nth0(IdxR,World,ColR),
    (IdxS is IdxR-1;IdxS is IdxR+1).
```

At first we only had support for directives such as *take* and *move* and later we added similar rules for the questions: *count*, *where* and *what*.

4.2. Quantifiers

The quantifiers allow the robot to handle query such has "*put any ball in the red box*". The quantifier function uses cuts to choose one possible action, when several are possibility are available to the robot (for example, `any([a,b])`). A cut (!) in prolog is a way to restrict the search by "fixing" the variables when a rule is determined to be true.
As an example, say we have the variables:

```
L1 = [1,2,3].
L2 = [3,2,1].
```

And the rules.

```
bar(X,Y,L1,L2) :- member(X,L1), member(Y,L2).
bas(X,Y,L1,L2) :- member(X,L1),!,member(Y,L2).
bac(X,Y,L1,L2) :- member(X,L1),!,member(Y,L2),!.
```

For *bar* we get the output:

```
(X = 1, Y=1); (X = 2, Y=1); (X = 3, Y=1);
(X = 1, Y=2); (X = 2, Y=2); (X = 3, Y=2);
(X = 1, Y=3); (X = 2, Y=3); (X = 3, Y=3).
```

However for *bas* we instead get, since X is fixed to 1 in the first instance:

```
(X = 1, Y=1); (X = 1, Y=2); (X = 1, Y=2).
```

And finally *bac*:

```
(X = 1, Y=1).
```

Below are a few examples of the results by the cuts performed by the function *handleQuantifiers*:

Original query	Interpreted query
`movebeside([a,b],[c,d])`	**`movebeside([a,b],[c,d])`**
`movebeside(any([a,b]),any([c,d]))`	**`movebeside([a],[c])`**
`movebeside([a,b],any([c,d]))`	**`movebeside([a,b],[c]))`**
`movebeside(any([a,b]),[c,d])`	**`movebeside([a],[c,d])`**

5. Planner

5.1. Terminal cases

The first step in our implementation was to implement terminal case, i.e. the move of an object that does not require more than two actions (pick up or drop). A terminal case could be “Put the black ball beside the yellow box“ if there is nothing on top of the black ball and nothing on the left of the yellow box.

The planner takes a query as an input and build list of triplets, where each triplet represent a feasible action. There are 3 possible terminal moves, each represented by a specific triplet:

Motion	Triplet
Pick	**[K_1,-1,move]**
Drop	**[-1, K_2,move]**
Move	**[K_1,K_2,move]**

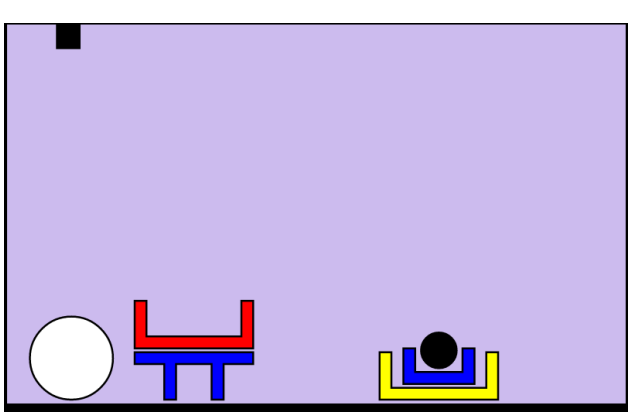
Where K_1 is the position of the object to pick and K_2 the position where the object has to be drop.

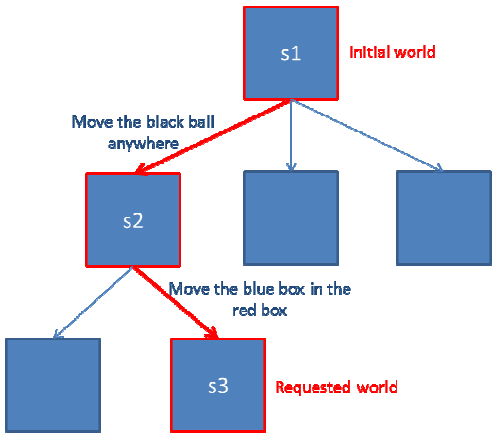
5.2. Complex cases

Once the terminal moves have been implemented, we can proceed to the implementation on more complex cases. Such as moving an object inside a box if there is already something inside this box.
The output of our plan function for complex cases is a list of triplets/actions (pick, drop, and move) that leads to the Goal.

In order to create this list of actions, the plan function is called recursively. The list of actions to reach the Goal is then a concatenation of all the actions to get from the original world to the requested world. This can also be viewed as a concatenation of all the branches of the decision tree used by prolog to find a solution to the requested state of the world (see below).

Here is an example:
The user query is “Put the blue box in the red box”.
The plan function performs an in-depth research in the prolog decision tree.





So the output of the plan looks like:
Plan = [[4,3,move],[4,2,move]]

5.3. Heuristic

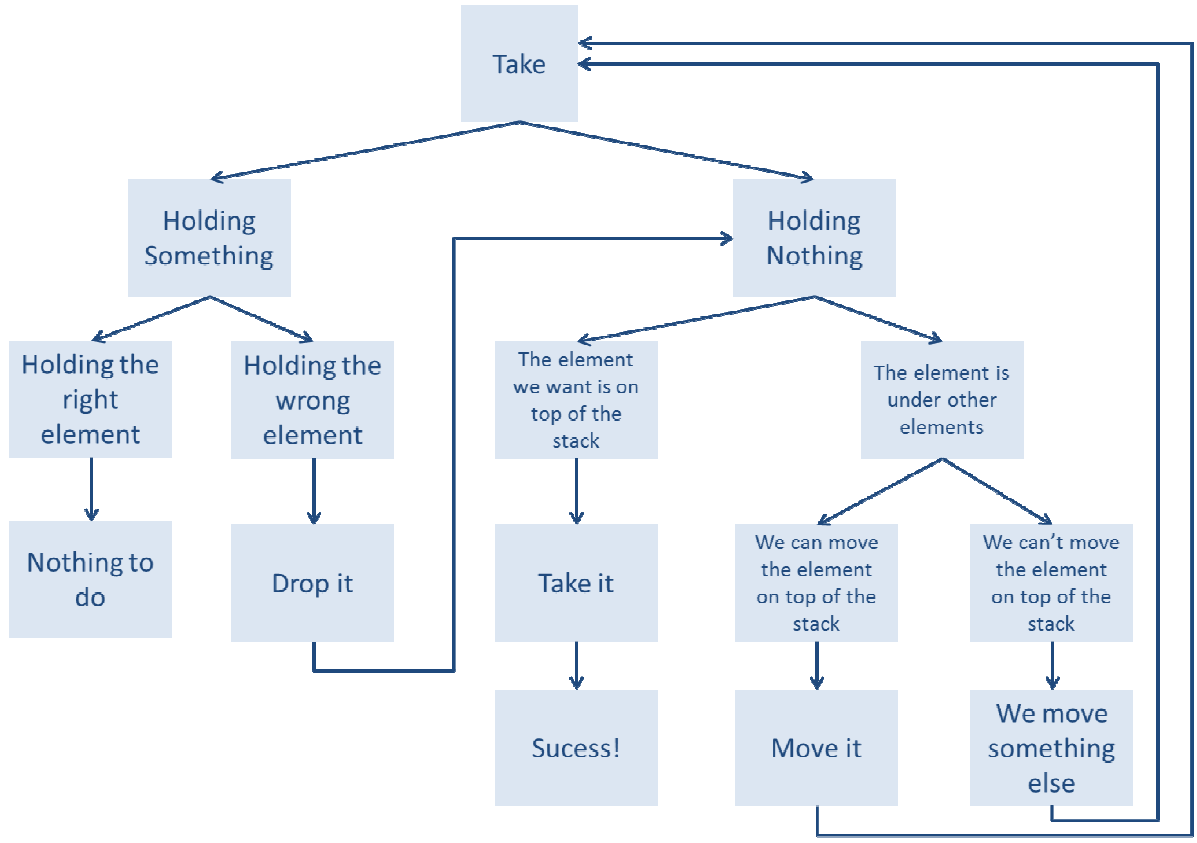
When the robot has to deal with not straight forward case, such as moving objects that are not on top of a stack or moving a ball above a table, we need the heuristic in order to optimize the way we handle these cases and to move objects in a better way.

The first heuristic we implemented was really simple. Basically, we added a rule allowing the solver to move any object in the world and our solver was doing a depth first search and simply checking that it was not looping on an already encountered world’s state. So it was just trying to go as far as possible in a branch of the decision tree, checking if it was meeting the requirements of the query and, if not, backtracking to the last node for which we did not explore all the rules that can be applied to this world's state.

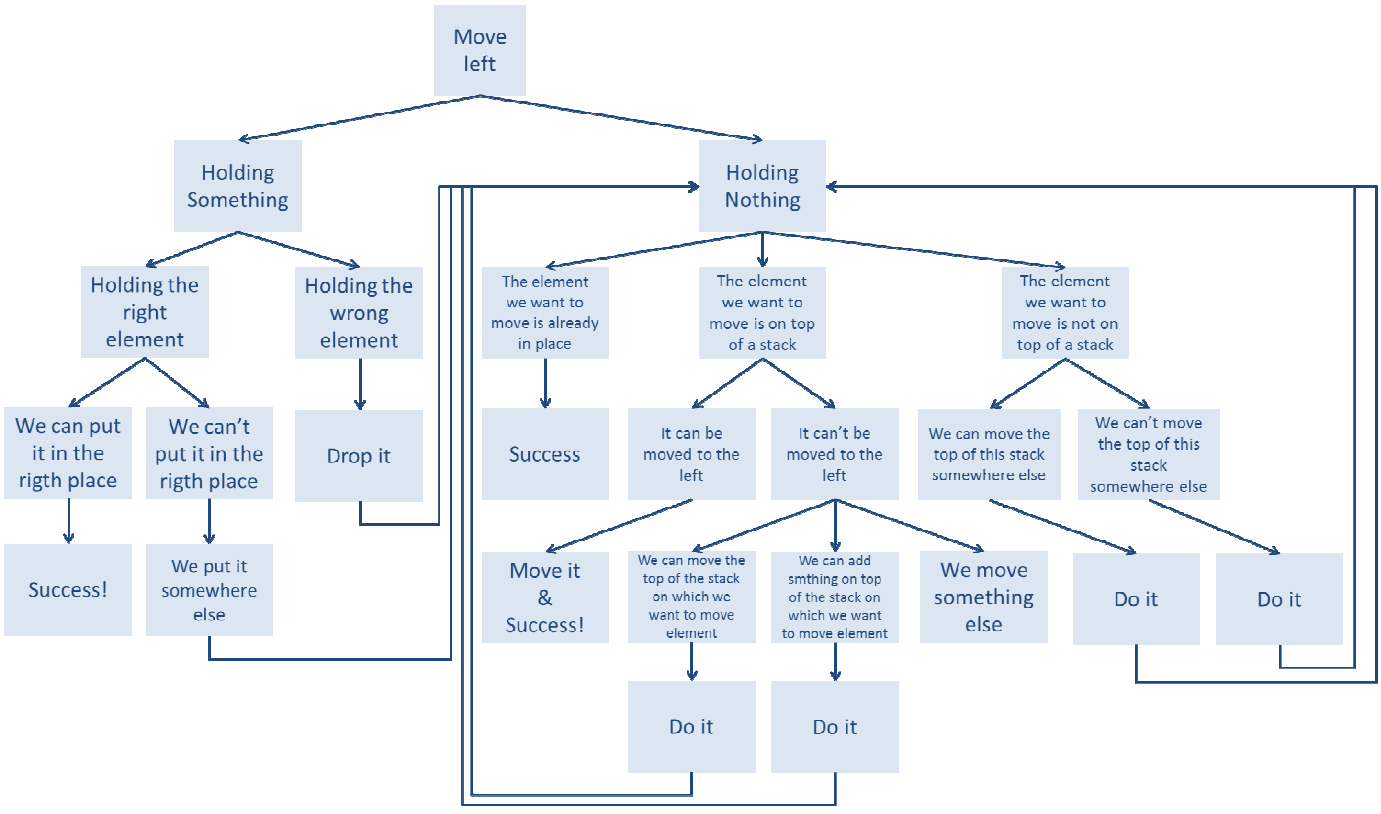
To be sure that the robot is not looping (if the world's state reached by the action of a rule has not been encountered before), we are initializing a list of the already reached world's states to the list containing only the original state and every time we apply a new rule we add to this list the newly reached state. To avoid looping we only allow to apply rules which do not produce an state already present in this list.

The new heuristic we implemented allows the robot to handle complex cases in a smarter way than the basic implementation we used.

For example, for the action “take”, the heuristic function as explain in the below scheme:



And for the move left we have the following heuristic:



6. Ambiguities handling

The ambiguities handling allow the robot to handle cases where the request from the user is not clear enough. For example, in picture 1, “what is under the box?” would lead to an ambiguity, since the robot would not know if the user want to know what is under the red box, the blue box or the yellow box.

In case of ambiguity, the robot asks the user for a precision. The user then has to precise the object he is referring to. In our example, it could be “the small blue box”.

Once the user has precised the object, the robot will get this information and try to match with the entire possible goals it has identified. It then selects the unique matching solution (if it exists).

If ambiguities still occur the robot then return an error. There is no second question asked to the user since prolog does not handle the while loop.

7. Output

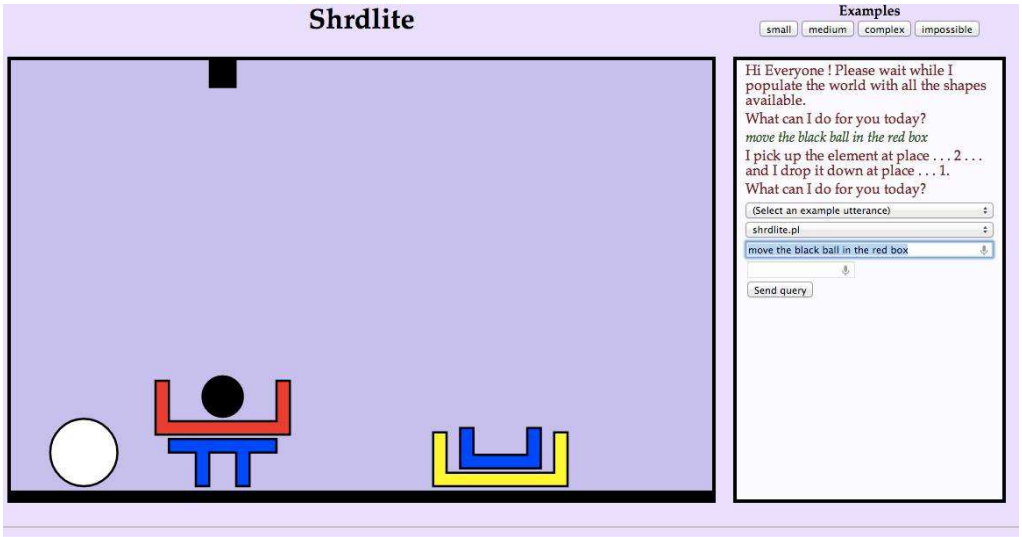
The output is a web interface developed in html and javascript. The world and all actions are represented with the SVG (Scalable Vector Graphics) library. It's a lightweight library for manipulating and animating vector object.

Once the heuristic is complete, the web client picks up results by means of an ajax request. The goal and the plan are already defined and execute in a console and we just execute it with javascript and SVG.

In case we have an ambiguity, we return a question that asks the user for more precisions. For example if we have the following query: "put the ball in the box" and there are two boxes, we ask for a precision, to which the user answers in the form "the white one".

Furthermore, the particularity of our robot is that possible outputs are not only actions (such as moving objects around) but it's might also be some verbal information such as the number of elements stack on top of each other, the number of squares, etc…

Below, you can see the representation of the world (on the left) and the interface to communicate on the right. The black square at the top represents the arm which takes object and moves to another place with an animation to follow steps of the problem's resolution.



8. Measuring efficiency

We did some test runs using the timer in linux as :

/usr/bin/time swipl -q -g main,halt -s javaprolog/shrdlite.pl

Unfortunately we saw that we used approximately the same memory and/or time regardless of the complexity of the world or problem, so the results are unfortunately not relevant for these tests. We present the raw data for the output.

```
"utterance": ["move", "the", "blue", "box", "inside", "the", "yellow", "box" ],

  "trees": [

    "move(basic_entity(the,object(box,-
,blue)),relative(inside,basic_entity(the,object(box,-,yellow))))"

  ],

  "goals": ["moveinside([m],[k])" ],

  "plan": [

    ["pick", 9 ],

    ["drop", 1 ],

    ["pick", 7 ],

    ["drop", 2 ],

    ["pick", 7 ],

    ["drop", 3 ],

    ["pick", 7 ],

    ["drop", 5 ],

    ["pick", 9 ],

    ["drop", 7 ]

  ],

  "output":"I pick up the element at place . . . 9 . . . and I drop it down at
place . . . 7. I pick up the element at place . . . 7 . . . and I drop it down at
place . . . 5. I pick up the element at place . . . 7 . . . and I drop it down at
place . . . 3. I pick up the element at place . . . 7 . . . and I drop it down at
place . . . 2. I pick up the element at place . . . 9 . . . and I drop it down at
place . . . 1. "

}0.06user 0.00system 0:00.06elapsed 98%CPU (0avgtext+0avgdata 5264maxresident)k

0inputs+0outputs (0major+1419minor)pagefaults 0swaps
```

```
{

  "utterance": ["move", "the", "blue", "box", "above", "the", "yellow", "pyramid"
],

  "trees": [

    "move(basic_entity(the,object(box,-
,blue)),relative(above,basic_entity(the,object(pyramid,-,yellow))))"

  ],

  "goals": ["moveabove([m],[i])" ],

  "plan": [

    ["pick", 4 ],

    ["drop", 1 ],

    ["pick", 2 ],

    ["drop", 3 ],

    ["pick", 4 ],

    ["drop", 2 ]

  ],

  "output":"I pick up the element at place . . . 4 . . . and I drop it down at
place . . . 2. I pick up the element at place . . . 2 . . . and I drop it down at
place . . . 3. I pick up the element at place . . . 4 . . . and I drop it down at
place . . . 1. "

}0.06user 0.00system 0:00.06elapsed 100%CPU (0avgtext+0avgdata 5276maxresident)k

0inputs+0outputs (0major+1421minor)pagefaults 0swaps
```

```
{

  "utterance": ["move", "the", "blue", "box", "beside", "the", "yellow", "box" ],

  "trees": [

    "move(basic_entity(the,object(box,-
,blue)),relative(beside,basic_entity(the,object(box,-,yellow))))"

  ],

  "goals": ["movebeside([m],[k])" ],

  "plan": [ ["pick", 3 ],  ["drop", 4 ] ],

  "output":"I pick up the element at place . . . 3 . . . and I drop it down at
place . . . 4. "

}0.05user 0.00system 0:00.06elapsed 100%CPU (0avgtext+0avgdata 5180maxresident)k

0inputs+0outputs (0major+1418minor)pagefaults 0swaps
```

9. Contribution of Pierre BOUTRY to the project

During this project I worked on several layers of the application. I started by writing the planner starting by the basic cases, then the complex cases and finally adding a smart heuristic to improve the treatment of robot.

I fixed a few minor bugs on the interpreter, based on the work done by Dan DOLONIUS on this part.

I added the possibility to define medium size object in JSON. This allows creating more complex and *realistic* world.

I implemented together with Julien MICHELET, the solver layer of the application. The function of the solver is to return a list understandable in Javascript. This list will be executed in Javascript on client side to perform the graphic animations of the robot as well as the verbal answers of the robot.

I also added the voice recognition brick to the application, so that the user can use a mic rather than a keyboard to send his query to the robot.

Finally, I work on how to handle “inversed list” on prolog side, since JSON and prolog seem to read list in an inversed way.

10. Contribution of Dan DOLONIUS to the project

During this project, I was in charge of writing and implementing the interpreter layer, which is in charge of taking the output of the parser and translating it to a query understandable by the planner. The interpreter has been modified several time during the project, either to handle more complicated cases or to allow the user to use quantifiers like “any”, “all” or “the”.

11. Contribution of Julien MICHELET the project

My first task on this project was to implement the improvement of the basic grammar. I added the possibility to ask new questions to the robot such as “what is under this object”, “where is this object” or “count the number of this specific object on this stack”.

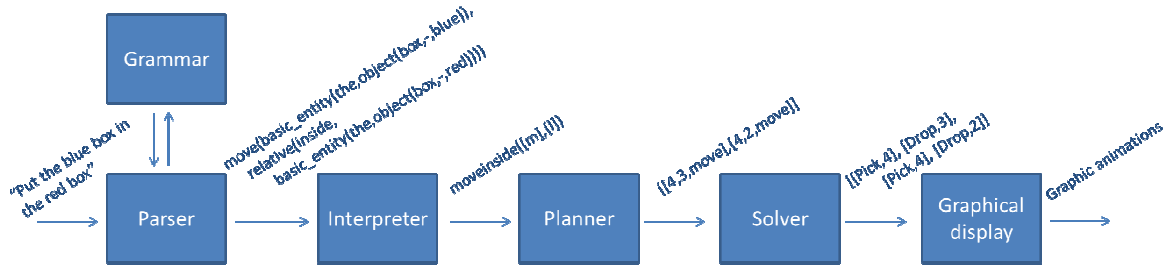
Once it has been done, I had to improve the interpreter as well as the planner so they can handle this new grammar and be able to answer the new type of questions offered to the user. I also implemented the last version of the canBeOn function. This function is used to check if an object can be put on top of a stack (possibly empty) and is used in the planner.

I implemented with Pierre BOUTRY the solver layer of the application. The function of the solver is to return a list understandable in Javascript. This list will be executed in Javascript on client side to perform the graphic animations of the robot as well as the verbal answers of the robot.

Finally, I implemented the ambiguities handling. For that I added the possibility for the robot to ask the user for complementary information regarding his query and added the proper treatment of this query.

12. Contribution of Emeric ROVERC'H to the project

For this project, I first try to define and present a clear macro presentation of the project. This would be used to present our project as well as facilitate the communication and interaction of the different people working on different parts of the project. I created the global application scheme below:



Then, once we had the first working version of the application, I started to gather the information from the teams working on each parts of the application. All the gathered information had then to be reformulated to be understandable by people outside of the project. Moreover, this allowed the team to stand back on the ongoing project and to refocus the efforts on the most important points of the project, rather to persist on none crucial features/actions.

In parallel with those reporting/macro tasks, I also helped the teams (in charge of distinct parts of the application) that were sometimes lacking of time/manpower to stay on track with the schedule we had defined.

For example, I helped Pierre BOUTRY with the implementation of the *smart* heuristic that has been presented in the report above.

I also finished to write some prolog function such has canBeOn, which is used to check if an object can be put on top of a stack (possibly empty), or RetrieveGoalElements, which allow to get from the interpreter output what object we want to interact with, what action we want to perform and we the object should be placed, using what has been implemented by the other member of the group.

I also performed the benchmark of the different implementations we have created during this entire project. This benchmark was based on the use of different worlds (small, medium and complex) as well as different heuristic we have implemented.