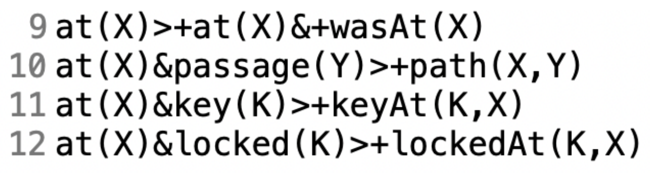
Assignment 3: Goal-Oriented Planning Agent

GROUP 43: MAKSYM LYTOVKA (S3705609), IVAN BANNY (S3647951)

Q1:

We define the percept rules as follows:

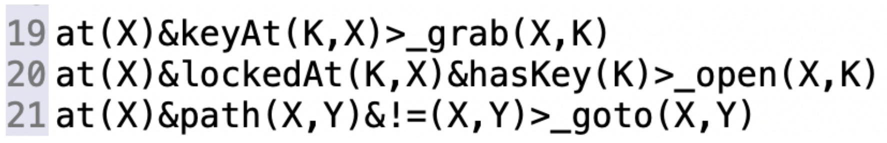


```
at(X)>+at(X)&+wasAt(X)
at(X)&passage(Y)>+path(X,Y)
at(X)&key(K)>+keyAt(K,X)
at(X)&locked(K)>+lockedAt(K,X)
```

When we perceive that we are in a certain position, we remember this fact, as we will use it for reasoning a forward-chaining. Also, we define the predicate wasAt(X). This will be extremely helpful in the future when we will adopt goals. We transform the predicate passage(X) with one predicate into the path(X,Y) (there is a path from X to Y), which will be helpful in the planning stage. Rememberring the position and type of the key is also more useful, than just knowing that the key is somewhere in the maze (key(K) was transformed into keyAt(K,X)). Finally, we define lockedAt(K,X), to store information about the doors. It follows the same logic, as keyAt(K,X) predicate.

Q2:

We define the program rules as follows:

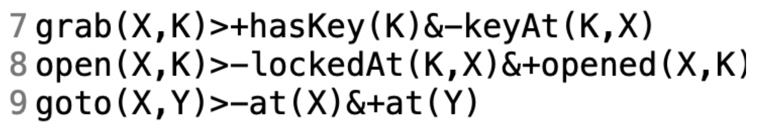


```
at(X)&keyAt(K,X)>_grab(X,K)
at(X)&lockedAt(K,X)&hasKey(K)>_open(X,K)
at(X)&path(X,Y)&!=(X,Y)>_goto(X,Y)
```

They are pretty straightforward. If you are in some position and the key is in the same position, you can grab that key. If you are in some position and the door is locked at that position and you have a key for that door, you can open the door. Finally, if you are at some position X and there is a path to some position Y, you can move there. Interesting to note, that following this rule, we can't have the path for not adjacent positions (X and Y have to be next to each other), as due to the rules defined for the world (Maze.java), the agent can't move more than one step at the time. However, more on that in section Q12.

Q3:

The action rules are defined as follows:

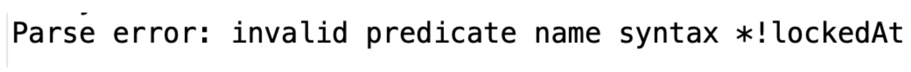


```
grab(X,K)>+hasKey(K)&-keyAt(K,X)
open(X,K)>-lockedAt(K,X)&+opened(X,K)
goto(X,Y)>-at(X)&+at(Y)
```

If we grab a key, we delete knowledge about the “key on the ground” from the belief base and add knowledge of possessing a certain key. If we open the door, we delete the knowledge about the locked door and add the fact, that the door was opened. The second fact will be needed for the inference and planning. We could not follow the rule:

open(X,K)>-lockedAt(K,X)&+!lockedAt(K,X)

because that is not proper syntax in terms of sfol. We got such an error:



```
Parse error: invalid predicate name syntax *!lockedAt
```

Finally, when we go from one place to another, we change our coordinate at(X). One could say, that there is no reason to assign at(Y), as we will get it from percepts. However, they would be wrong, as we need to do planning, in which there will be no surroundings and we will update our belief base only using action rules.

Q5:

MyAgent.unifiesWith(…) has 2 arguments: Predicate p (predicate) and Predicate f (fact). With the assumption, that f is fully bound, the algorithm for unification simplifies significantly. The algorithm is the following:

1. Check, that the predicate f is indeed bound (if not - return null, otherwise continue)
2. Check whether the numbers of terms in two predicates are the same (if not - return null, otherwise continue)
3. Check whether the names of the predicates are the same (if not - return null, otherwise continue)
4. Go through every pair of terms in p and f:
    1. If term in the p is a constant:
        1. If not equal to the term in the f - return null
        2. If equal - continue with step 4 (for loop)
    2. If a term in the p is a variable:
        1. If this variable is not in the substitution - add it to the substitution and continue with step 4
        2. If it is already in the substitution and the values of the substituted constant are different - return null, otherwise continue with step 4

Q6:

The main idea of the algorithm in the MyAgent.findAllSubstitutions(…) method, is to traverse a tree looking for proper substitutions. Each state of the tree consists of the already done substitutions (HashMap<String, String> substitution in the code) and the conditions left (Vector<Predicate> conditions) to substitute. The root node contains no substitutions and all the conditions passed by the call of the function. The return of the method is true if we find at least one. The base case for our search is if there are no more conditions to find substitutions, that’s the leaf node. In that case, we may simply add the found substitution to the general list of found substitutions.

If we are not in the base case, we pick the last condition from the vector and consider it. And here we have two options: either that’s a reserved predicate, or not. And these two choices are the answer to the question of how we deal with the reserved predicates. Firstly, let’s consider the case, if we encounter not reserved predicate. For this case, we go through the whole KB we have, trying to unify the predicate with some facts from the KB. If we succeed in that and newly found substitutions do not contradict the previously found:

```
if (unification != null && unionIsPossible(substitution, unification))
```

we go deeper. Hereby, the number of children of the node is the number of successful unifications for the last condition in the list (vector).

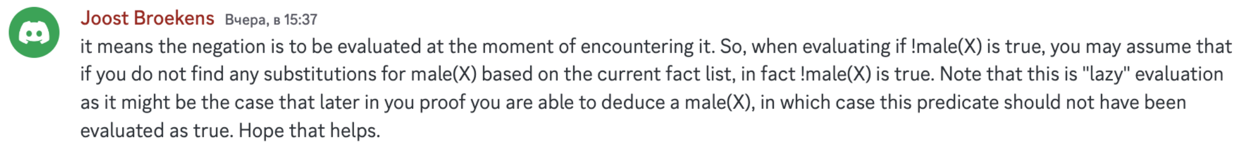
If we encounter a reserved predicate, we try to move it to the beginning of our list (the end of our “queue”). However, if there are only reserved predicated left (we check it using MyAgent.onlyReservedPredicatesLeft(…) helper method), we have to deal with that. And dealing with them is quite simple: we just have to check, if they’re true. If they are false, we return null, otherwise continue. We can use such a strategy, as we moved all the reserved predicates to the back, meaning that the variables in them must be already in the substitution we consider.

Q7:

For the MyAgent.forwardChain(…) the main idea of the algorithm is simple: we take KB and go through every rule in that KB, unifying conditions with the facts known and trying to get the new facts (conclusions) until there are no new facts found on the present step. In our code, we first divide all the sentences in our KB into rules, facts and actions, as that will facilitate a further loop. Afterwards, until no new facts are found, we loop through every rule and apply MyAgent.findAllSubstituions(…) on that. If there are some substitutions, we apply them to the conclusion of the rule and add these facts (or actions) to the list of new facts (or new actions). After that, we check whether the inferred facts are new, by checking whether the new facts set is the subset of old facts and whether the new actions set is the subset of old actions. The helper method MyAgent.isSubet(…) was developed for such a check. After the loop has finished, compose the KB again, from the decomposed set of rules and actions. As the resulting KB has to have only bound sentences, we omit the rules set.

Q8:

As for our code, the negation operation was handled together with all the other reserved predicates (but not actions) in the MyAgent.findAllSubstitutions(…) method. If we encounter it, we check for two conditions: if the same predicate (negated) is present in the KB or if there is no not negated predicate in the KB, then negation is true. In case of any of them, we assign the predicate to be true. We hope, that that’s what Joost meant in his message in Discord (sorry for the white theme):


Joost Broekens Вчера, в 15:37
it means the negation is to be evaluated at the moment of encountering it. So, when evaluating if !male(X) is true, you may assume that if you do not find any substitutions for male(X) based on the current fact list, in fact !male(X) is true. Note that this is "lazy" evaluation as it might be the case that later in you proof you are able to deduce a male(X), in which case this predicate should not have been evaluated as true. Hope that helps.

As for the MyAgent.unifiesWith(…) method, we see no need to handle reserved predicates there, as we do the check for the match of the names of the predicates.

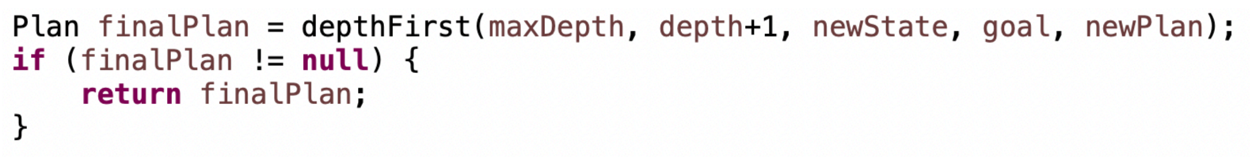
Q10:

The code for the MyAgent.idSearch(…) is trivial. This method is just a wrapper for the MyAgent.depthFirst(…). The algorithm goes as follows:

1. Initiate the depth to 1
2. Try to find the plan via the MyAgent.depthFirst(…) with the given depth. If a plan was found, return it. Increase the depth by 1 otherwise.
3. Repeat step 2 until the depth becomes higher than the max depth.
4. If no plan has been found so far - return null.

Q11:

Now we consider the most important algorithm in the section of planning: depth-first search. The algorithm traverses the tree of states, looking for a state, in which the goal is satisfied. As a return, it yields a list of actions, needed to get from the root node to the node with the goal. If there is no satisfactory node, the algorithm returns null. In our algorithm, there are two base cases: when we reached the max depth of the search or when we satisfied the goal. In these cases, we return the null of the plan found respectively. On the other hand, if we are not in the base case, we need to continue the search. We do that by creating a new imaginary agent and performing actions on him. To be more precise, we create this agent and perform forward chaining (think(…)) on the KB provided to the algorithm. After that, we have a set of intentions. Then, we try every action possible from the intentions base (apply act(…)). This way we arrive at the children of the state. If any of the successors found a plan, we return it (push it up to the root, in some sense). We do that in the following lines:

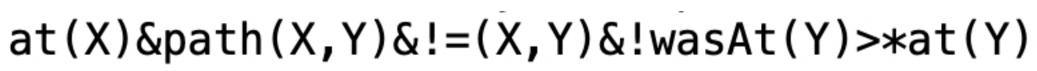


```
Plan finalPlan = depthFirst(maxDepth, depth+1, newState, goal, newPlan);
if (finalPlan != null) {
    return finalPlan;
}
```

We decided to call the variable the finalPlan, as the plan we get from the algorithm was produced by reaching the base case. Otherwise, we would get a null. Finally, if all the children of the root node return null, we return null for the whole algorithm.

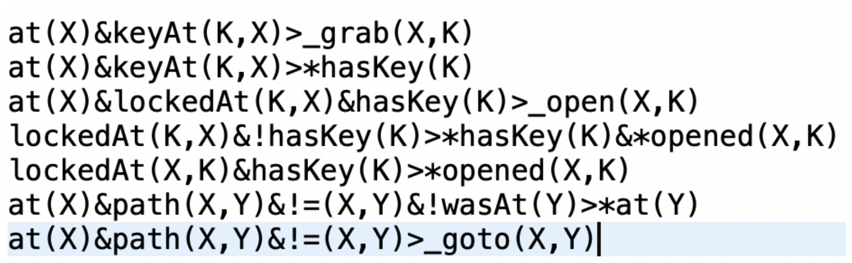
Q12:

The main problem we encounter at that moment is the priority of the goals of the agent. In section Q1, we introduced the predicate wasAt(X), which indicates to which locations the agent has already been. In the step 9., we added the following rule:



```
at(X)&path(X,Y)&!=(X,Y)&!wasAt(Y)>*at(Y)
```

This rule adds the goal of visiting some unexplored positions to the agent. It goes after all the other goal-adopting rules in the program.txt:



```
at(X)&keyAt(K,X)>_grab(X,K)
at(X)&keyAt(K,X)>*hasKey(K)
at(X)&lockedAt(K,X)&hasKey(K)>_open(X,K)
lockedAt(K,X)&!hasKey(K)>*hasKey(K)&*opened(X,K)
lockedAt(X,K)&hasKey(K)>*opened(X,K)
at(X)&path(X,Y)&!=(X,Y)&!wasAt(Y)>*at(Y)
at(X)&path(X,Y)&!=(X,Y)>_goto(X,Y)
```

So we first add the goals about picking the key or opening the door. We do so to facilitate the execution of the actions in logical order. For instance, imagine, that you are in the cell with the key. If you adopt the goal of exploring first, you will explore first. In other words, you will make a step to another cell, and only afterwards will you come back to the previous cell and pick up the key. Obviously, it's better to get the key first. That's why the order matters.

However, there is a problem with this order. Imagine, that you are stepping into the cell with a key from another cell. Then, you add a goal of picking the key. But there were goals of exploration already in the goals KB. That's why you go out of the cell, go to the other side of the map, get to this cell, and only afterwards come back (through the whole map again) and pick up the key. It sounds like a very not optimal plan. That's why, the possible tweak to tremendously optimize the search is to implement the goals KB as a stack and not as a queue. So, the last goal we add is the first one we want to achieve. That may be done by getting not the 1st sentence in the KB, but the last one.

Another possible tweak is related to the surroundings of the agent. At the present moment, it doesn't allow the agent to step more than one cell. And that's understandable. It's not that fast to check whether there is a path from one cell to another. Nevertheless, such a feature of a maze could reduce the cost of planning. Because we could write a rule of:

```
path(X,Y)&path(Y,Z)&!=(X,Z)>path(X,Z)
```

Which would get us all the routes our agent can "come up with". Hereby, we would not need all the actions like goTo(1_1, 1_2), goTo(1_2, 1_3) and goTo(1_3, 1_4), we could simply reason with goTo(1_1, 1_4) and cut all these steps.