Assignment 2: A simple reflexive agent in Prolog

Joost Broekens

October 11, 2022

1 INTRODUCTION

In this assignment you develop a simple reactive agent in Prolog. Prolog is a logical programming language. It enables you to write down facts and rules in the form of predicate logic. These facts and rules together form the "program". However, this program is not executed in the typical way. Prolog sees these rules and facts as a database against which it will run a query. The query is a question that needs to be proven. The database contains facts and rules that can be used to proof whether or not the question is true. During this exercise we will get more experienced in programming in Prolog. The goal is not to become a Prolog expert, but to better understand the process of *proof generation* and the role of *substitution*, *unification* and *search strategy* on this process. Prolog's default search strategy is backwards chaining. When you get stuck with Prolog, a good source of information is the following website:
http://www.learnprolognow.org/lpnpage.php?pageid=online

First download and install SWI prolog from:
https://www.swi-prolog.org/download/stable

The assignment is graded, and you should submit it through your group assignment in Brightspace in the following format: a zip file including your two code files (wumpus.pl and family.pl), and a report in pdf format.

2 THE FAMILY

Ok, let's start. First create a new file called "family.pl" in your favorite editor, and copy the content from family.pl downloadable from Brightspace (or just download it directly). Make sure you save the file in the default Prolog file folder (e.g., Documents/Prolog on Windows). That way you are sure Prolog will find the file in its path when you want to use it. Let us assume a typical family of brother's sisters, parents, etc.. Such facts about the family are defined in the form of Prolog facts. A fact can be interpreted as a predicate. Predicates have no, one, or even more terms. Investigate the following Prolog code defining a family.

```
%A typical family database.
male(joost).
male(sacha).
male(leon).
male(merlijn).
male(peter).
female(sofie).
female(sandrine).
parent(joost,sacha).
parent(joost,leon).
parent(sandrine,sacha).
parent(sandrine,leon).
parent(fien,sofie).
parent(fien,merlijn).
parent(peter,fien).
parent(peter,joost).
```

The first couple of facts (predicates) tell us something about people, for example, there is a $joost$ for whom $male(joost)$ is true. People are represented as simple terms. The latter facts tell us about relations between terms, for example that $joost$ is the $parent$ of $sacha$. Notice that Prolog statements always end with a period (.).

If we want to verify if, e.g., joost is the father of sacha, then we can ask Prolog to try to proof the following statement:
$parent(joost, sacha)$. (note the period at the end).

To type and run a query against a database (your program). First you tell Prolog to load your program by typing $[family]$. in the Prolog command prompt (the window that opens when you start Prolog). The you type the query in the Prolog command prompt, and press enter. It is important to remember that every Prolog query, fact, and rule always ends with a period (.).

If we want to know who are the children of joost, then we can ask Prolog to give us all values for which variable X unifies with the database (i.e., all values of X that would proof the following $parent(joost, X)$.. Note that variables always start with a capital!

If there are multiple answers to a query, you can get all answers to the query by pushing the *n* key in the console. If you want to stop a query, hit <ctrl>-c then hit the 'a' key (abort).

It is now time to study the topics we covered on search, predicate logic, and inference with predicate logic. If you have not done so, please do this first.

2.1 Questions

For the following questions we assume the family database above.

1. Execute the query $parent(joost, X)$., and explain the results by showing step by step how the substitution and unification works.

Now let's consider how we can tell Prolog some more advanced stuff. Rules! Rules consist of predicates as well. In fact a Prolog rule is a *first order definite clause*. So, the mathematical form of a rule is:
$condition(term_1) \wedge condition(term_2) \wedge ... \rightarrow conclusion(something)$

However, in Prolog we write this as follows:

```
conclusion(something):-condition(term_1), condition(term_2).
```

We can then add the conditions as facts:

```
condition(term_1).
condition(term_2).
conclusion(something):-condition(term_1), condition(term_2).
```

With these facts added, you could ask Prolog to proof $conclusion(something)$ and it would succeed.

You can also add variables to rules. An example with our family database would be:

```
isChild(X):-parent(Y,X).
```

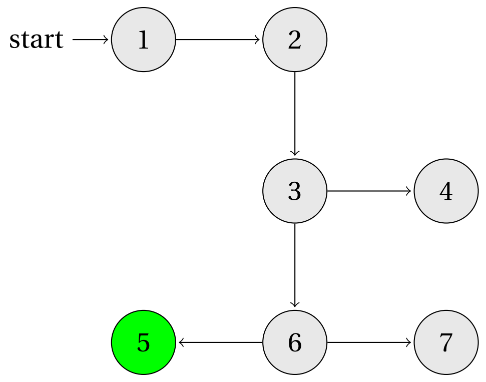
This defines the rule for deciding if the predicate isChild(X) is true.

2. What is the output of the query isChild(X)? Explain the results again using unification and substitution.
3. Define the predicate brother(X,Y) based on already existing predicates, with X being the brother of Y. Do the same for sister. Think about unification of X and Y very carefully! Give proof that the rules are correct and complete, by showing the full list of brothers and sisters (remember to get all answers to the query). Explain the double entries.
4. There is one important fact missing, which you will be able to find while looking at the answers to your brother and sister query. What is this fact? Also add it to the database (your program).

5. Define the predicate cousin(X,Y), with X being the cousin of Y (in Dutch: "neef van jouw generatie, dus een zoon van jouw oom of tante"). Use your brother/2 and sister/2 (/2 meaning with 2 terms) predicates.
6. Define the predicate family(X,Y), which should be true when X and Y are blood-related family. Hint: make use of a new recursive predicate ancestor(X,Y) defining that X is an ancestor of Y. Hint two: to do this all, you need multiple rules for the same predicate, i.e., there are more definite clauses needed per predicate to be able to derive the correct proof of being a blood relative.
7. Is Sandrine family of sofie? Execute a query to find out and explain your result by showing what rules are tried in a step by step manner indicating both unification and substitution steps.
8. What is the query for finding the list of family members of Sandrine. Do the query, and explain the result by showing what rules are tried in a step by step manner indicating both unification and substitution steps.
9. Explain the result of family(leon,peter), by showing what rules are tried in a step by step manner indicating both unification and substitution steps.
10. Explain the result of family(sofie,sacha), by showing what rules are tried in a step by step manner indicating both unification and substitution steps.

3 A SIMPLE REFLEXIVE AGENT

Now that we have a good understanding of prolog, we will move to the second part of this assignment. The part that has to do with agent building. Assume that there is an agent that needs to find a goal (let's say some food) in a maze (a very simple one, that is). Let's assume the maze is depicted below.



This maze is represented as a directed acyclic graph, i.e., a graph without cycles. The robot starts in node 1, and the food is in node 5, along the way it needs to make choices but can never go back. It is important that you represent this structure in Prolog first. So, open a new file in your editor and call it "wumpus.pl". To get you started, use the following code. It defines that location "1" is where the robot is, and location "5" where the food is. It further gives you the first correct link predicate to indicate an edge between location 1 and 2.

```
%The location of robot and goal
robot(1).
goal(5).

%The first link as a predicate
link(1,2).
```

3.1 QUESTIONS

First we have to build a model of the environment in the form of facts about links between locations.

11. Define the rest of the links so that the structure represents the graph.

Now that we have the model of the environment as facts in the database, we move towards the acting part of the agent. For this assignment we stay in the Prolog environment. Therefore we will "simulate" the agent's actions by executing the moves the agent suggests ourselves. To be able to actually move the agent, we need to be able to remove and add facts about *robot* from the database, namely its current location. This is done by adding the following code to the top of your program (do this now!):

```
:- retractall(robot(_)).
:-dynamic robot/1.
```

This tells Prolog to do two things when it loads this database: first remove all predicates $robot(\_)$, where (_) means with any term; second, define the $robot(\_)$ predicates as a dynamic one with one term, meaning you are now allowed to add and remove facts about robot, such as where it is. To add or remove a fact, you can use $assertz(\_)$ and $retract(\_)$ with any fact you want. For example, the code to add any link from X to Y could be done as follows (don't do it, it's just an example):

```
addlink(X,Y):- assertz(link(X,Y)).
```

To execute a single move, we need to define the predicate $move(L) : -....$ Of course a robot can only move to neighbouring locations that are immediately $adjacent$. If the robot smells food in an adjacent location, it will suggest that one. So we need to define $suggest(L) : -...$ giving us this location, if we are next to it. For example, $suggest(5)$ is true if $robot(6), goal(5)$.

So the agent's perception, reasoning, action cycle is as follows: the agent sees that there are adjacent other nodes, and it smells the food if it is close. Based on that, it has a reflex: either propose any action, or, suggest the food location. You will have to execute the action by asking the $move(n)$ query.

12. Define the predicate $adjacent(L)$ and $move(L)$, and play around with it by executing the query in the console and testing if the robot moves correctly. Explain your predicate.

13. Define the predicate $suggest(L)$, and explain why it is correct and complete. Would it work if you have more than one goal?

4 TOWARDS A PLANNING AGENT

Now that we have a simple reflexive agent that checks and executes the moves you tell it to make, and suggests the best move when it smells the food, we are getting somewhere. However, this agent is rather "myopic", it only looks at surrounding spaces, while it has a *model* of the world already available to it. Let's find out how we can make the agent plan a route to the goal and suggest a more useful next location rather than fail on that when the goal is not adjacent to it. First make sure that the program resets to $robot(1)$ and $goal(5)$.

4.1 QUESTIONS

14. Define a predicate $path :- ...$ that is true if there is a path from the robot to the goal. You should use basic recursive search making use of the *directed* structure of the graph. There is no need to construct, return, or move along the path yet, just proof if it exists.

Ok, we have a way to proof if there is a path from the agent to the food. But, we also want to know what the path looks like. So in a way, we want to force Prolog to, while proofing if a path exists, unify the elements of the proof into a *list*. The way to deal with lists in Prolog is straightforward but a little difficult to get used to. Look at the code example below (don't add it to your code).

```
list([H|T],H,T).
```

If we ask Prolog the following query $list([1,2,3],X,Y)$. then this will result in $X = 1$ and $Y = [2,3]$. This is because if $[H|T] = [1,2,3]$ and H needs to unify with the head of the list and T with the tail, then $H = 1$ and $T = [2,3]$, hence $X = 1, Y = [2,3]$. The brackets tell Prolog you are talking about a list, and the pipe symbol in the list [|] simply means: what is left of it is the head, and what is right of it is the tail.

We can do useful things with this, such as concatenating lists to form a new list. See the following code:

```
append([], X, X).
append([X | Y], Z, [X | W]) :- append(Y, Z, W).
```

This code defines two predicates for $append$, the first is simple, it is true if the first list is empty and the second and the third are equal. This must be the case, because appending an empty list to some list X should result in that same list X. The second is the real magic: it is true if *the head of the resulting list and the head of the first list are equal and the rest Y of the to be appended list $[X|Y]$ can be proven to also be appended to W*.

15. Just to get you a bit familiar with lists, add the last piece of code to your program. Why does the append predicate work? Show the step by step unification process for $append([1,2],[3,4],X)$.

16. Now adapt your path predicate to not only proof that there is a path to the goal location, but also return that path as a list of next nodes to visit, i.e, the predicate $path(P) :- ...$

should be set up such that if $robot(1), goal(5)$ then $path([2,3,6,5])$ should be true :-). Note that you don't need to make use of the $append(...)$ predicate, but you do need to make use of lists.

17. Finally, use your new path predicate to make a better version of suggest, such that it will always suggest the next location towards the goal.

This all works quite nicely, but to show you that there are simple limits to what this type of "planning" by proof can do, consider the following. What would happen if the world looked a little bit different? For example, there is a way back from from node 3 to node 1?

18. Add the fact $link(3,1)$ to the top of your other link facts (so directly above them). Now execute the query $suggest(X)$, and wait for the result. It does not come. Explain why, in detail. Use Prolog's default search strategy, which is backwards chaining, substitution, unification, and importance of the order of rules and facts in your answer. See the relevant section in the book (Chapter 9 in particular) to help you.

This concludes this assignment. Make sure you answer all questions in a report (pdf), and you upload both your code and your report in one zip file at the group assignment in Brightspace.