Assignment Week 1: Prolog basics

Birna van Riemsdijk

Introduction

The objective of the practical assignment of this week is to introduce you to the logic programming language Prolog. More background information about the language can be found in the online book Learn Prolog Now[1]. In MOD8 you will be introduced to the language in more detail, and learn skills to write more elaborate programs.

In this course the main purpose of the practical assignment is:

- to apply your theoretical understanding about logic and reasoning for simple practical knowledge representation problems in Prolog
- to compare your theoretical knowledge about logic and resolution theorem proving to its implementation in Prolog

The main purpose of the practical assignment of this week is to install Prolog, run a basic program, and familiarize yourself with the basic aspects of the programming environment. The assignment of this week is **not** handed in or graded, however, **it is strongly recommended to make this assignment as a preparation for Week 2**. Experience has shown that familiarizing oneself with a different programming paradigm, in particular logic programming, needs time for digesting the material and learning a new way of thinking.

Assignment

Some practical tips on the use of Prolog are at the bottom of this document, which may be useful to check if you run into problems or have questions while making these exercises.

Exercise 1: Getting started

1. In order to try and test your Prolog code, please download the SWI prolog programming language from `http://www.swi-prolog.org/`.
2. Create a file with a basic text editor and the extension `.pl`, for example `test.pl`.
3. A Prolog program consists of logical formulas written in the syntax of Prolog, which can also be referred to as a Knowledge Base (KB). Any statement (formula) in Prolog is ended with a full stop. The most basic statements are atomic propositions, which are called *facts* and which always start with a *lowercase letter*. In the file `test.pl`, write the Prolog program containing only the atomic proposition 'p', i.e., `p.` with full stop is the text of your first Prolog program.
4. Follow practical assignment 1.4[2] of Learn Prolog Now to load your program `test.pl` into Prolog. Note that commands for loading knowledge bases also end with a full stop!

[1]`https://lpn.swi-prolog.org`
[2]`https://lpn.swi-prolog.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse4`

Exercise 2: Horn Clauses

Not all (propositional) logic formulas can be represented in Prolog: statements are restricted to so-called *Horn Clauses*.

1. Horn Clauses can be facts, as we have seen in the assignment above, or they can be so-called *rules*. Rules are implications such as $p \wedge q \Rightarrow r$, which in Prolog are written with the implication symbol reversed, i.e., $r \Leftarrow p \wedge q$, and using the Prolog syntax for conjunction (comma) and implication (`:-`). That is, in Prolog this implication is written as `r :- p,q.`. This implication is a Horn Clause because we can rewrite it as $p \wedge q \Rightarrow r \equiv \neg(p \wedge q) \vee r \equiv \neg p \vee \neg q \vee r$, which is a disjunction of literals with (at most) one positive literal, namely r. That is, rules in Prolog have a head (`r` in this example) which is a single positive literal, and a conjunction of (positive) literals as the body.

   Modify your knowledge base to add this rule and reload it in Prolog. You can use the `listing.` statement to check that your new rule has been added.

2. As a preparation for the next step, add the clause `:- dynamic q/0.` at the top of your knowledge base and reload it (see Practical Tips below for an explanation why this is needed).

3. Computation in Prolog is done by asking the knowledge base whether certain formulas follow from it. This is called a *query*. The so-called *inference engine* then performs resolution in order to determine whether the formula represented in the query follows from the knowledge base or not. Queries can only be performed for atomic formulas (or conjunctions), i.e., we cannot ask Prolog whether a rule follows from the knowledge base.

   Ask Prolog whether r follows from the knowledge base by typing the query `r.` after the Prolog prompt (`?- r.`) and pressing 'return'. What is Prolog's answer? Now perform a proof by resolution of whether r follows from $p \wedge q \Rightarrow r$ and p, i.e., whether $p \wedge q \Rightarrow r, p \vdash r$ (as in Exercise 3 of the tutorial exercises). If you did it right, you should get the same answer!

4. Now add the proposition q to the knowledge base as fact `q.` and reload it. Perform the query `r.` again. What is the result? Can you reproduce it using resolution?

Exercise 3: Backward Chaining

The inference mechanism of Prolog is called *backward chaining* because in Prolog we start with what we want to derive, i.e., the goal (the query r in the assignment above), and then Prolog consults the knowledge base to see which rules need to be applied in order to derive it.[3]

1. To understand the idea of "chaining" the application of rules, add the fact `b.` and the rules `a :- b,c.` and `c :- d.` to your knowledge base, and the statement `:- dynamic d/0.` at the top of the file as above. Reload your program. Now ask Prolog whether a follows from this knowledge base by posing this as a query. Prolog should answer 'false', i.e., a does not follow.

2. One can think about this inference as follows: in order to derive a, Prolog needs to apply the rule `a :- b,c.` (since it has a as the conclusion, i.e., if b and c hold then a follows), and in order to apply this rule, Prolog has to be able to derive in turn b and c. These become the new query goals, and the same inference mechanism is performed again. Prolog tries to derive b, which it can do because that is a fact in the knowledge base. Now it has to derive c, which it could in turn do by applying the rule `c :- d.` (if d can be derived then we can conclude c). Now d becomes the new goal, however, d cannot be derived since it is neither a fact in the knowledge base nor the conclusion of any of the rules. Therefore this derivation fails and Prolog returns 'false'.

[3]Alternatively, *forward chaining* would take all the facts in the knowledge base and apply all rules that it can to derive new facts.

Now add the fact `d.` to the knowledge base and pose the query `a.` again. Now Prolog should return 'true' because the rule `c :- d.` can be applied to derive c. In the background, these applications of rules are in fact steps in the resolution proof. To practice with resolution, you can see if you can prove that a indeed follows from the formulas $b, d, d \Rightarrow c, b \wedge c \Rightarrow a$. Can you match the steps in your proof with the application of rules in Prolog?

Exercise 4: Wumpus World

In this exercise you will program small parts of the knowledge base that an agent would need for the Wumpus World. The Wumpus world is introduced during the lecture and in the book by Russel & Norvig (Section 7.2).

1. Specify a rule in Prolog that will allow the agent to conclude that there is gold in square [2,3] based on information from its sensors about the presence of glitter, i.e., the rule should specify that if there is glitter then there is gold in this square. Use atomic propositions `gold23` and `glitter23` to define your rule. Reload your program, and pose the query `gold23.`. Does it succeed? If not, what is needed to let it succeed?

2. Add facts that specify there is no pit and no wumpus in square [1,1]. This information is given due to the setup of the Wumpus world: [1,1] is the starting position of the agent and it does not encounter a Wumpus or pit there since otherwise it would loose the game immediately. Prolog does not allow to specify explicit negation (Prolog offers an alternative, but we will not go into that in this assignment); instead add facts `nopit11` and `nowumpus11`.

3. Write a rule that will allow the agent to conclude that square [1,1] is safe (`safe11`) from the information that there is no pit and no Wumpus at this square.

4. The agent can conclude that [1,1] is safe because it knows from the start there never is a Wumpus or pit in the starting square. However, to conclude that other squares are safe, it needs to use and reason with information that it gets from its sensors while standing on neighbouring squares. In particular, it will sense a breeze if it is standing next to a pit, and a stench if it is standing next to the Wumpus.

   Write two rules that allow the agent to conclude that squares [1,2] and [2,1] are safe, based on information the agent receives from its sensors at square [1,1] that there is no breeze and no stench (`nobreeze11` and `nostench11`). Reload your program. Pose a query to ask Prolog whether squares [1,2] and [2,1] are safe. The query should not succeed, since we only have the rule, but no information from its sensors yet.

5. Simulate the reception of sensor information by the agent by posing the queries `assert(nobreeze11).` and `assert(nostench11).` in the Prolog prompt, which adds corresponding facts to the knowledge base after the program has been loaded (see practical tips below). Pose again a query to ask Prolog whether squares [1,2] and [2,1] are safe. Do you notice a different result?

6. This captures the first reasoning steps that the agent needs to do in the Wumpus world discussed during the lecture. You can experiment with Prolog by trying to program the next reasoning steps, e.g., to conclude that there are possible pits in squares [2,2] and [3,1]. You might quickly notice that programming the agent in this way is quite cumbersome as we have to define the reasoning steps separately for each square in the Wumpus world. Next week you will learn about predicate logic, which introduces variables in the logic (and correpondingly these can be used in Prolog) which will make such programming much easier. Nevertheless seeing how far you get and which challenges you encounter can be a good exercise!

Practical Tips

A few practical tips about usage of Prolog are below:

- Use **comments** (% ⟨put your comment here⟩ or `/*` ⟨put your comment here⟩ `*/`) to explain your code.
- If you want to abort a query, e.g., when you receive an error message, type the letter ‘a’.
- If in your rules you want to use atomic predicates such as `q` that are not (yet) specified as facts or occur as the head of a rule in the knowledge base, add a statement such as `:- dynamic q/0.` at the top of your Prolog program. This tells Prolog that facts of this kind may be added later, and prevents an “undefined procedure” error (ERROR: r/0: Undefined procedure: q/0) when posing a query that uses the corresponding rule. The error occurs because in order to derive r, Prolog needs to try to derive q, however for q there are no facts or rules (with q as the head) in the knowledge base.
- To add and remove a fact such as `p` to the knowledge base used by Prolog after your program has been loaded, the queries `assert(p).` and `retract(p).` can be used from the Prolog prompt. This can be useful to experiment with dynamic changes in knowledge, for example to model changes in the agent’s environment over time and receipt of information from its sensors, without having to modify your program with each change. Note that these facts remain in the knowledge base even after reloading your program, which means you need to explicitly retract any facts that you no longer need. Use the query `listing.` to check whether the fact has been successfully added or removed.
- Use the query `halt.` to stop Prolog.