Practical Assignment Week 2

Birna van Riemsdijk

Introduction

Important: before starting on this assignment, first make the practical assignment of Week 1. That assignment introduces you to the language Prolog, which is necessary knowledge to be able to successfully make this assignment.

This practical assignment is made using 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

For this assignment you will need to write a number of Prolog clauses and hand in working Prolog files that implement each of the assignments correctly, as well as explain aspects of unification and proof search in Prolog.

Assignment

Make sure that you adhere to the following requirements. Failing to adhere to these can be expected to result in an *insufficient grade*:

- You need to make sure that all code you write can be **executed**. Please submit one Prolog file (".pl" files) on Canvas as your answer.
- Always use **comments** (% ⟨put your comment here⟩ or `/*` ⟨put your comment here⟩ `*/`) to explain your code.
- The assignment should be made in **pairs**. Copying from other pairs is not acceptable.

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.

[1] `http://www.learnprolognow.org`

Getting started

This part of the assignment does not need to be handed in, but introduces you to more expressive features of Prolog that correspond with predicate logic as introduced during the lecture. In particular, it shows you how to use predicates with arguments, variables and unification in Prolog.

1. Just like in predicate logic, in Prolog we can make use of predicates (atomic sentences) that have a number of terms as *arguments*. For example, add the facts `p(a).` and `p(b).` to your knowledge base and load the program. In Prolog, predicate names (`p` in this example) and constants which can form arguments to predicates (`a` and `b`) start with a *lower case* letter.

2. Predicates are not only defined by their name, but also by their so-called *arity*, i.e., their number of arguments. In Prolog the arity of a predicate is denoted with a slash and the number, e.g., `p/1` to denote the predicate p with one argument. We can define another predicate with the same name but a different number of arguments, e.g., `p(X,Y)` which can be referred to as `p/2`. This notation can show up in warnings or exception messages of Prolog, and it is used for defining dynamic predicates (see practical tips at the bottom of this document).

3. Just like for propositions without arguments, we can ask Prolog by means of a *query* whether a predicate with arguments follows from the knowledge base. For example, pose the query `p(a).` to Prolog by typing it in the prompt `?-` and pressing return. Prolog returns 'true', since this occurs as a fact in the knowledge base and therefore follows from the knowledge base.

4. We can also use *variables in queries*, to ask Prolog to find a unification for this formula such that if applied to the formula it follows from the knowledge base. Variables in Prolog start with an *upper case* letter. To see how this works, pose the query `p(X).` to Prolog. Prolog now returns the first unification, namely `X = a` for which this formula follows from the knowledge base. That is, if we apply this unification to `p(X)`, we get `p(a)` which indeed follows from the knowledge base.

   We can ask Prolog to try to *find another unification* by typing semicolon ';' after it has returned its first unification. Now Prolog returns `X = b`, since we also have `p(b).` in the knowledge base. Since Prolog has already established there are no other possible unifications, it now shows the prompt again.

5. We can also use *variables in rules* to define more complex predicates than we could do with only propositional logic atoms. For example, we can specify 'if `p(X)` then `q(X)`' by adding the rule `q(X) :- p(X).` to the knowledge base. In Prolog we do not have quantifiers, but rules should be read as implicitly universally quantified, i.e., this rule corresponds to the predicate logic formula $\forall X(p(X) \Rightarrow q(X))$. With this rule we can now derive new knowledge from the facts `p(a)` and `p(b)`. To show this, add the rule, reload your program, and pose the query `q(a).` to Prolog to ask whether this follows from the knowledge base. Prolog should now answer 'true'.

6. Similarly as for queries regarding facts in the knowledge base, we can use *variables in queries* to ask Prolog to find unifications for formulas that are the *conclusion of a rule*. To see this, pose the query `q(X).`. Prolog now answers `X = a`, i.e., `q(a)` follows because in order to derive `q(X)` for some `X` Prolog tries to derive `p(X)` by using the corresponding rule, and it can indeed derive it with this unification. We can again ask Prolog to find another unification by using the semicolon. It now also returns `X = b`.

Exercise 1: Proof Search

In this exercise we will use the trace functionality of Prolog. By means of trace, we can see exactly which steps Prolog performs for trying to prove a query goal. This means that we can also see in which order Prolog applies the rules and tries to prove subgoals.

The mechanism through which Prolog decides which rules to apply and which subgoals to prove can be viewed as a search strategy. That is, Prolog searches for a proof to show that a certain query goal follows from the knowledge base, starting with the toplevel goal that the user specifies in the prompt and trying out relevant rules one by one to see if the goal can be derived. By using the trace functionality, we can thus see which search strategy is implemented in Prolog. We will examine Prolog's search strategy in this exercise.

Note that the trace functionality can also be useful for debugging purposes, which you may use in the other exercises.

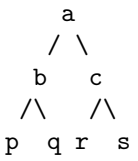
1. Consider the Prolog program below. Load this program into Prolog and activate the trace functionality using `trace.` (Deactivating tracing is done using the query `notrace.`)

   ```
   p.
   q.
   r.
   s.

   a :- b,c.
   b :- p,q.
   c :- r,s.
   ```

   A proof search in Prolog can be thought of as traversing a search tree where each of the nodes in the tree is an atom that Prolog tries to prove.

   Pose the query `a.` while having tracing activated. The first line will look something like this: `Call: (6) a ?`, where 'Call' means that Prolog starts to try to prove that a follows, which corresponds with expanding the node a. A similar line with 'Exit' means that Prolog has succeeded in proving a. Use 'return' to ask Prolog to go to the next step in the proof search (Prolog shows 'creep' to denote it has moved on to the next step of the search process), and keep doing so until the search finishes and Prolog shows the prompt again. The search tree for the program above and the query `a.` is depicted below.

   In which order were nodes of this tree expanded to find a proof for this query? Provide a list of atoms as your answer, e.g., `a,r,s,p,c,b,q`. *Write your answer in comments in your Prolog file!*

2. Does Prolog perform a depth-first search or a breadth-first search? Explain your answer. In which order would Prolog traverse the tree if the other of these two search strategies was used? *Write your answer in comments in your Prolog file!*

Exercise 2: Corona Regulations

Important note: the corona regulations described in these assignments are incomplete and likely out of date by the time you make this assignment. Therefore only use the corona regulations described here for creating your Prolog code, and not for deciding how to act in your own daily life yourself!

In this exercise you will implement a number of corona regulations in Prolog that express how to behave in order to follow these regulations. The resulting knowledge base may be used for

example for an agent that can support people in following the corona rules. In this exercise we focus on regulations for when to stay home and when to wear a mask.

Part I: Staying Home

The purpose of this assignment is to write a number of Prolog rules that define the predicate `stay_home(Person)`, which specify when `Person` should stay home. Ensure that the predicate returns 'false' for queries with someone that is not part of the list of persons as defined by the corresponding predicate `person` in the knowledge base, e.g.,. `stay_home(chang).` should return 'false'. First add the following facts to your knowledge base:

```
person(alisha).
person(frank).
person(eve).
person(alex).

corona_symptom(cold).
corona_symptom(cough).
corona_symptom(trouble_breathing).
corona_symptom(fever).
corona_symptom(nosmell).

has_corona(frank).

has_symptom(alisha,cold).
has_symptom(alex,stomach_ache).

housemate(frank,eve).
```

As an example, the following Prolog rule can be used to specify that a person should stay home if they have corona.

```
stay_home(Person) :-
     person(Person),
     has_corona(Person).
```

Check that the predicate returns 'true' for the query `stay_home(frank).`. You can also use a variable in the query, e.g., `stay_home(P).` to ask Prolog to find all unifications for `P` such that the formula follows, i.e., all persons that have to stay home. After defining this rule, Prolog should only return `P = frank`.

1. Specify a Prolog rule for the predicate `stay_home(Person)` saying that `Person` should stay home if they have a corona symptom, as defined through the predicate `corona_symptom(Symptom)`. For example, Prolog should return 'true' for the query `stay_home(alisha).`.
2. Specify one or more Prolog rules for the predicate `stay_home(Person)` saying that `Person` should stay home if they are a housemate of someone with corona, as defined through the predicate `housemate(Person1, Person2)`. For example, Prolog should return 'true' for the query `stay_home(eve).`.

Part II: Wearing a Mask

The purpose of this assignment is to write a number of Prolog rules that define whether someone should wear a mask while doing a certain activity. Ensure that the predicate returns 'false' for queries with someone that is not part of the list of persons as defined by the corresponding predicate `person` in the knowledge base, and similarly for activities not specified as such through the predicate `activity`. First add the following facts to your knowledge base:

```
person(mary).
person(john).
person(tabitha).

public_building(store).
public_building(museum).
public_building(municipality).
public_building(station).

activity(working).
activity(commuting).
activity(shopping).
activity(cycling).
activity(theatre).
activity(dance).

visit(store, shopping).
visit(station, commuting).

contact_profession(physiotherapy).
contact_profession(pedicure).

profession(mary, physiotherapy).
profession(john, pedicure).
profession(tabitha, scientist).

personal_mask_exception(john).
activity_mask_exception(theatre).
activity_mask_exception(dance).
```

3. Specify a Prolog rule for the predicate `wear_mask(Activity,Person)` saying that `Person` should wear a mask when the `Activity` requires that they visit a public building. Use the corresponding predicates `visit(Building, Activity)` and `public_building(Building)` to determine which building needs to be visited while doing the specified activity and whether this is a public building. Prolog should for example return 'true' for the query `wear_mask(shopping,mary).`, but 'false' for `wear_mask(cycling,mary).`. You may introduce an auxiliary (=additional) predicate `wear_mask(Activity)`, since this regulation does not depend on the specific person.

4. Specify a Prolog rule for the predicate `wear_mask(working,Person)` saying that `Person` should wear a mask when they are working in a contact profession. Use the corresponding predicates `profession(Person,Profession)` and `contact_profession(Profession)` to determine the profession of the person and whether this is a contact profession. Prolog should for example return 'true' for the query `wear_mask(working,john).`, but 'false' for `wear_mask(working,tabitha).`.

5. The corona regulations specify rules for when a mask should be worn, but also define a number of exceptions to these rules. Two of these exceptions are when a person cannot wear a mask, e.g., due to other physical health issues, or when the performed activity is granted an exception, such as creating theatre or dancing.

   Specify one or more Prolog rules for the predicate `no_mask(Activity, Person)` saying that `Person` does not have to wear a mask while they are doing `Activity` if they have a personal mask exception or if they are performing an exempted activity. Use the corresponding predicates `personal_mask_exception(Person)` and `activity_mask_exception(Activity)` to

determine the respective exceptions. Prolog should for example return 'true' for the query `no_mask(commuting,john).`, but 'false' for `no_mask(working,tabitha).`. You may specify two separate Prolog rules for each of these cases, or use the Prolog syntax for disjunction, a semicolon ';', in your rule.

6. You may have noticed that specifying regulations on mask wearing and their exceptions in this way can give conflicting results, for example for the queries `wear_mask(working,john).` and `no_mask(working,john).`, since both return 'true', i.e., when working john has to both wear a mask and not wear a mask. How could this issue be solved, i.e., what is needed to deal with regulations and their exceptions in a way that does not give such conflicting results? *Describe your answer in words in comments in the Prolog file.* You do not have to write the corresponding Prolog code.

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 `p` 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 p/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 when posing a query that uses the corresponding rule.
- 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.