COMP24412
Academic Session: 2022-23

Lab 2: Prolog
Assessed exercises

Joe and Giles

Submission is via git for this lab - see the section at the end of this document about submission. You should use your `comp24412` repository (branch `lab2`) as a starting point. Please let me know if running `git checkout lab2` and then `./refresh` doesn't work. You should see a file called `database.pl`.

These exercises concern the Prolog programming language, which is based on first-order logic. In Prolog, we work in a *fragment* of first-order logic: we are restricted to 'Horn Clauses', which means clauses in which at most one literal is positive. In Prolog, a Horn Clause is written in the form

```
g :- p1, p2, p3, ..., pN
```

Where `g` and `p1, ... , pN` are atoms. You should think of `:-` logically as meaning an implication pointing from right to left, so the expression above means the Horn Clause

$g \vee \neg p1 \vee \neg p2 \vee \neg p3 \vee ... \vee \neg pN$

To an extent, you can view a Prolog exercise as a logical modelling exercise with the syntactic restriction that you can only use Horn Clauses in your axioms. However, Prolog adds a couple of features from outside of first-order logic: in this lab we are interested in lists and arithmetic. You can read about Prolog from a practical point of view here: https://lpn.swi-prolog.org/lpnpage.php?pageid=online. In this course, we cover material corresponding to the first 5 chapters, but for the lab, you only need to look at Sections 1.2 (which defines syntax, e.g. how we write variables, predicate symbols, etc), 4.1 (which defines the syntax for lists), and 5.1 and 5.2 (which describe how arithmetic works).

Prolog has some features which it is better to ignore for the lab. If you know about Prolog already, you are strongly advised **not** to use negation or 'cut'. The automated marking will assume these are not used, and marks lost due to this assumption are your responsibility! (For the experts: it is ok to use negation if in your solution it behaves like logical negation, and cut if it does not affect the logical meaning of the knowledge base; a so-called 'green cut').

Learning Objectives

At the end of this lab you should be able to:

1. Use the Prolog interactive mode to load and query Prolog programs
2. Model simple constraint solving problems in Prolog
3. Define recursive relations in Prolog

Part 0: Getting Started

There are no mark available for the tasks in Part 0, but you will probably find them helpful before starting the assessed exercises. In fact, you may find it useful to work through a few of the formative exercises, described in the other document provided, even though they carry no marks.

Running Prolog (read this)

Open up a terminal and run `swipl`. You have just started the SWI-Prolog interactive mode, it should look a bit like this.

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 5.7.11)
Copyright (c) 1990-2009 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

The `?-` is the prompt where we can type things. **To get out of interactive mode you should type `halt.` (the `.` is necessary).**

The expected usage is that your Prolog program (knowledge base) is stored in a file and you load this into the interactive mode and query it. For the warmup exercises we suggest you create a file `warmup.pl` (open it in gedit or similar) and run `swipl` in the same directory. Alternatively, we can add queries directly to a program and run this from the command line using the `pl` command but we assume that you are using the interactive mode.

Warning. Not all Prolog implementations are the same. We are using SWI-Prolog in this course because it is what is on the teaching machines. They also have Sictus Prolog if you prefer to use that.

To load your `warmup.pl` file into interactive mode type

```
1 ?- [warmup].
true.
```

The `true` response tells you that the file was loaded correctly. You can now type queries directly in the interactive mode. Add the line `loves(mouse,cheese)` and `loves(monkey,banana)` to `warmup.pl` and run the following query

```
2 ?- loves(X,Y).
X = mouse,
Y = cheese
```

(You probably need to reload the file). Recall that this returns an *answer substitution* (assignment to variables). Type ; to get more answers or . to stop searching. In this example what happens to the answers you get if you swap the two lines you have in `warmup.pl`?

If you change `warmup.pl` you can reload it into interactive mode by typing `[warmup]` again. You don't need to stop and restart interactive mode.

The rest of this document contains the assessed exercises. Formative exercises in more of a tutorial style continue in the other document provided.

Part 1: Electrical Interference

You have joined a team working on the space program of a small country. The team is working hard trying to design a probe to send to a distant planet. The electrical engineering department is working on a list of components of the probe. For each pair of components, they have done tests to see whether the components can be put close to each other without interfering with each other and breaking. They have summarised their results so far in the following Prolog database (there is a copy of this in `database.pl` in the `lab2` branch of your git repo.).

```
component(antenna).
component(transponder).
component(radar).
component(spectrometer).
component(imu).
component(camera).
component(cpu).
component(ram).

safe_with(radar,cpu).
safe_with(cpu,radar).
safe_with(radar,imu).
safe_with(imu,radar).
safe_with(imu,camera).
safe_with(camera,imu).
safe_with(imu,cpu).
safe_with(cpu,imu).
safe_with(imu,ram).
safe_with(ram,imu).
safe_with(ram,cpu).
safe_with(cpu,ram).
safe_with(ram,camera).
safe_with(camera,ram).
safe_with(camera,transponder).
safe_with(transponder,camera).
safe_with(camera,cpu).
safe_with(cpu,camera).
safe_with(cpu,spectrometer).
safe_with(spectrometer,cpu).
safe_with(cpu,antenna).
safe_with(antenna,cpu).
safe_with(antenna,spectrometer).
safe_with(spectrometer,antenna).
safe_with(antenna,transponder).
safe_with(transponder,antenna).
safe_with(transponder,spectrometer).
safe_with(spectrometer,transponder).
```

Your task is to write software in Prolog to help the designers of the probe check that their designs will work, in that they only place components close together if the engineers have shown those components don't interfere. You should create a file `electrical.pl` to work in (make sure it is added to the `lab2` branch of your repo!).

Exercise 1
To get started, write a predicate `safe_list/1` (i.e. a predicate with functor name `safe_list` and taking 1 argument) which takes a list of components and which succeeds if and only if every component in the list is safe for use with every other component. For example `safe_list([cpu,imu,radar])` should succeed, but `safe_list([ram,cpu,radar])` should fail. You may define extra predicates if you wish (this is true for all exercises in this lab). □

Now, the engineers don't just want to study lists of components, they also want to analyse abstract designs for the probe. A design is a list where each element is either of the form `part(C)` for some component `C`, or else of the form `shield(L)`, where `L` is a design and the first element of `L` is of the form `part(c)`. For example

```
[part(imu), shield([
part(cpu),shield([part(cpu)])
])
]
```

is a design, but

```
[part(imu), shield([
shield([part(cpu)]),shield([part(cpu)])
])
]
```

is not, because all the elements in the argument of the outermost `shield` are themselves of the form `shield(X)` – the first one is not of the form `part(X)` as required.

You can assume that you will be given a properly formatted design. However, you may find it instructive to write a predicate to check whether a design is properly formatted (no marks for this).

The meaning of `sheild(L)` is that the elements of `L` have been electrically shielded from the outside world. That means that we can check whether a design is safe by checking whether all the parts which are not inside shields are safe when used with each other, then recursively checking whether the contents of each shield are a safe design. For example, the design

```
[part(radar), part(imu), shield([
part(antenna),part(transponder)])
]
```

is safe, because the `radar` and `imu` are safe together, and the `antenna` and `transponder` are safe together. But

```
[part(radar), part(camera), shield([
part(antenna),part(transponder)])
]
```

is not safe because it puts the `radar` and `camera` together without surrounding one of them with shielding, while

```
[part(radar), part(imu), shield([
part(antenna),part(camera)])
]
```

is not safe because the `camera` and `antenna` are not safe together.

Exercise 2
Write a predicate `safe_design/1` which, when given a design as input, succeeds if and only if the design is safe in the above sense. Note it is unlikely that you will be able to use the predicate `safe_list/1` from Exercise 1 directly, however the structure of your answer to this exercise is likely to be similar. It is likely that you will need to define auxiliary predicates. You do not need to worry about how your predicate behaves if the argument is not a design according to the definition above. If you are struggling with this part, you might find the next one easier. □

Since the finished probe is supposed to be carried into space by a rocket, it is important for it to be as light as possible. The engineers are trying to use as little shielding as possible. As a first approximation, they want the software to count the number of times `shield` is used in a design.

Exercise 3
Write a predicate `count_shields/2` which succeeds if and only if its second argument is the number of times `shield` occurs in its first argument. It must be possible to use your predicate in the query `count_shields(D,N)` where D is a design and N is a *variable*, so that Prolog returns the number. For example

```
count_shields([part(imu),shield([
part(ram),shield([part(radar)])
]),
shield([part(cpu)])
],N)
```

should return `N = 3`. You do not need to worry about the behaviour of your predicate if the first argument is not a design according to the first definition. □

Finally, the engineers want to check that a design does not have any components missing. The idea is that they will provide a design and a list of components, and the software should be able to tell if each component in the list is used exactly once somewhere in the design.

Exercise 4
Write a predicate `design_uses/2` which accepts a design and a non-empty list of components as arguments, and succeeds if and only if the first argument is a design, and the components used in the design are exactly those listed in the list. Each element in the list must be used exactly once in the design (if a component is repeated, it should be occur as many times in the design as there are repetitions of it in the list). Note that the elements may appear in a different order in the design compared to the list. Finally, the predicate should fail if the first argument is not a design as defined above (we do count the empty list as a design).

Before attempting to write `design_uses/2`, you should first write a predicate `split_list/3` with three arguments, which succeeds if the first argument can be split up into the other two arguments. For example, the query `split_list([1,2,3],X,Y)` should return the results

```
X = [1, 2, 3],
Y = []

X = [1, 2],
Y = [3]

X = [1, 3],
```

```
Y = [2]

X = [1],
Y = [2, 3]

X = [2, 3],
Y = [1]

X = [2],
Y = [1, 3]

X = [3],
Y = [1, 2]

X = [],
Y = [1, 2, 3]
```

It may be helpful to think about going through the list element by element, and allocating each element either to the second or third argument, before recursively allocating the rest. You may then like to contemplate the results of the query `split_list([1,2,3],[H],R)`.

Make sure your test your solution(s). You should ensure that your solution works where components are repeated in the component list, for example, `design_uses([part(imu),shield([part(imu)])],[imu,imu])`. More testing data may be provided via Blackboard. Please keep an eye out for announcements.

Exercise 5
Once you have finished the exercises above, you should try running the query `design_uses(D,[imu,cpu])`. Ideally this will list all possible designs using one `imu` and one `cpu`. This will depend on ensuring your previous definitions are not unnecessarily restrictive. If this works, try running the query

```
design_uses(X,[transponder,spectrometer,antenna,ram,cpu,imu,radar,camera]),
safe_design(X),
count_shields(X,2)
```

although you might prefer to see if you can find a solution manually first.

It might be that your Prolog code goes into an infinite loop! If you're really interested in Prolog, ensure that your implementation of `design_uses\2` does *not* go into an infinite loop for the input given.

□

Submission and Mark Scheme

Submission is via `git`. You must work on the `lab2` branch and *tag* your submission as `lab2_solution`. You must push your tagged commit to Gitlab before the deadline.

Your work will be marked by a series of automated tests, designed to check each point in the mark scheme below thoroughly.

The marking scheme (10 marks total) is as follows.

- (1 mark) for correct `safe_list/1`
- (2 marks) for correct `safe_design/1`, 1 mark if missing a case
- (2 marks) for correct `count_shields/2`, 1 mark if missing a case
- (4 marks) with 2 marks for `split_list/2` and 2 marks for the final `design_uses/2`
- (1 mark) for an implementation of `design_uses/2` which does not go into an infinite loop.

You must provide clear comments which explain the idea of your code; in the event of technical issues with the automated tests, your work may be marked by hand, and you may lose marks if it is not clearly commented. This is particularly true for solutions to Exercise 5.