COMP24412
Academic Session: 2018-19

Lab Exercise 1: Getting to Grips with Prolog

For this lab exercise you should do all your work in your COMP24412/ex1 directory.

The deadline for this lab is exactly **1 week** after the start of your lab (16:00 on Tuesday for Lab Group H and 9:00 on Thursday for Lab Group F). We expect the lab exercise to take you 4-5 hours.

See the end of this document for details on the submission and marking process.

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
4. Explain issues surrounding termination in Prolog
5. Use the accumulator approach to detect cycles in reasoning

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.

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 a file (your Prolog 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
```

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.

Part 0: Warming Up (1 hour)

In this part you're going to write some simple Prolog programs and think about what they do. You don't need to submit anything (there are no marks for this). This part is just to get you warmed up. In general it is best to think about what should happen before you type things in. If you understand something then move on to the next thing. Y

Facts and Queries

Above you gave an extensional definition for a predicate `loves/2`. We write it as `loves/2` to indicate that it has the name `loves` and arity 2. You don't type in the 2. The reason we do this is because you're allowed to define a separate predicate `loves/3` etc. Try adding some more facts above this predicate and asking different queries of the facts. If your query contains variables then Prolog will *match* it against the known facts. Try and write some queries that just return `true`, some that return `false`, and some that return an answer substitution.

Experiment with Unification

Recall that two terms unify if we can apply the same substitution to both terms and produce two terms that are syntactically equal. In Prolog the notion of equality is to do with whether things unify.

Decide if the following terms are syntactically unifiable (without occurs-check) using the builtin `=/2` predicate on the prolog shell and write down your results. You can use `subsumes_term/2` to check if the first argument matches the second term (but if this is confusing then skip it for now).

Note that `=/2` is special as you can use it *infix*. So the normal `=(1+1,2)` can be written as `1+1 = 2`, which is a bit nicer to read. Just type these queries into interactive mode (they don't rely on any defined predicates).

term A	=	term B	A matches B	A unifies with B
1+1	=	2		
X+Y	=	2		
X+Y	=	Z		
`f(a)`	=	`f(a,b)`		
`f(X,Y)`	=	`f(Y,a)`		
`f(X,a)`	=	`f(b,X)`		
`f(X,Y,Z)`	=	`f(Y,Z,X)`		
`f(X,X,Y)`	=	`f(Y,a,b).`		
`f(X,X,Y)`	=	`f(Y,a,a).`		
`[X|Xs]`	=	`[A,B]`		
`[X,Y,Z]`	=	`[[A,B]|[C,D]]`		

Why does `1+1` not unify with —2—? Note that in `1+1` the + is infix and the term could be written `+(1,1)` now the equality `+(1,1) = 2` is no different to `f(a,a) = b` for the purposes of unification. Numbers and symbols such as + are just treated as any other symbol.

For the last case concerning lists you may want to write out the syntax tree for the two lists. If the list notation is confusing then recheck the lecture slides. As a quick reminder `[X | Xs]` stands for a list with head `X` and tail `Xs`.

Revisit `isa_list/1`, `member_of/2`, `nonmember_of/2`

Go through the slides from 11.2.2019 and extract the definitions of `isa_list/1`, `member_of/2`, `nonmember_of/` (or rewrite them on your own if you feel confident). For each predicate, write a query containing variables that succeeds and one that fails. All of the queries should terminate, ie. `query, false.` must return with `false`.

More pattern matching with Lists

Lists are important in Prolog. We will often need to pattern match against lists. For example, add following rule to `warmup.pl`

```
bigger_than_one([_,_|_]).
```

It defines a relation `bigger_than_one` that is true on all lists with at least two elements. It does this by matching against the start of the list with `_,_`, ensuring at least two elements, and then the rest of the list with `|_` allowing for longer lists. Try this rule out on various lists.

The rule

```
same_head([X|_],[X|_]).
```

relates any two lists with the same first element (head) regardless of length etc. Try it out on some pairs of lists, what about lists containing lists? Now

- Write a rule that relates any two lists with the same second elements.
- Write a rule that relates any two lists such that one is a prefix of the other e.g. they are equal up to the length of the shortest.

Define all different

Using `nonmember_of/2`, define `alldifferent(List)` such that all elements in List are different.
Hint: think about a suitable base case. Then define the recursive that describes what additional constraints have to hold if a new element is prepended to this list.

Try your predicate on the following queries:

```
alldifferent(Xs).
alldifferent([]).
alldifferent([a, A, B])
alldifferent([a | Xs])
alldifferent([X, Y, X | Zs])
Xs=[_,_,_,_,_], member_of(X,Xs), alldifferent([X|Xs])
```

Can you find some other interesting queries?

Exploring terms

The following is less important but interesting. If you're running out of time then consider skipping it, or just reading through it without doing the exercises, and moving onto Part 1.

Terms are built out of functions applied to objects but these are not functions in the sense of the mathematical functions you may be used to. They are term algebras or algebraic data types. For this reason we should, perhaps, call function symbols *constructors* as they construct new terms from smaller terms. In contrast to other programming languages that contain data-types, Prolog performs no type-checking. This is the reason for defining predicates in the style of `isa_list` that describe those terms that are lists.

To understand what this means let us first note that objects are really zero-arity functions (i.e. functions applied to zero arguments). Due to the *unique name assumption* we have that `a = b` is always false as objects with different names are always different.

The same concept applies to terms. Functions applied to different terms represent different objects e.g. because `a` and `b` are different the terms `f(a)` and `f(b)` are always different. The nice thing about this characterisation is that terms from an algebraic data type have a single model (all symbols are interpreted as themselves).

We use terms to store structured data. You saw this in one of the lectures where we defined a list using a `next` constructor. Another common data type you will use is that of a *tuple* - a fixed-length list of values. The semantics of terms mean that two tuples are always different if their contents is different. This is, therefore, a good way of encoding some state of a system or a row of a table.

A common data type used in many examples is that of *natural numbers*. This has one object (`zero`) and one constructor (`succ` for successor). Terms are of the form `zero`, `succ(zero)`, `succ(succ(zero))`. If we wanted to introduce the name `two` for `succ(succ(zero))` we might try and write `two = succ(succ(zero))` but this won't work because our notion of equality (=) is that of algebraic data types (which doesn't allow this). It is possible to define our own equality over terms but this usually needs the help of a predicate that can describe the structure of a term (a so-called meta-logical predicate) which has not been mentioned in the lecture yet.

To get some experience modelling things using terms, have a go at completing the following Prolog program by adding more students, courses, and groups and finishing the relations. You will need to

```
student(bob).
course(comp24412).
group(h).
entry(row(Student,Course,Group)) :-
  student(Student), course(Course), group(Group).

table(Table) :-
   % check that Table is a list of rows

select_table_group(Table,Group,Result) :-
   % check that the rows in Result are exactly
   % the rows in Table that have group Group
```

This kind of idea is similar to what you should be thinking about in Part 1.

Part 1: Describing Solutions

A major difference between Prolog and languages you may be more familiar with is that Prolog is a declarative programming language e.g. one describes what a solution to a problem looks like rather than how to produce it. We explore this idea in this part of the lab.

The problem is that of scheduling. I have to meet 6 project students in pairs and have 3 slots in which I can do this but there are some constraints. Let us call the students `student1` etc and the slots `slot1` etc and fix that a higher-numbered slot occurs after a lower-numbered slot. The constraints are that:

1. `student1` must meet in `slot1`
2. `student2` and `student3` must meet in the same slot
3. `student1` and `student4` cannot meet in the same slot
4. `student6` cannot meet in `slot1` or `slot3`

Clearly I should meet each student exactly once.

We suggest that you start by specifying which objects are students using a `student/1` relation. Remember that unless you explicitly mention an object it does not exist due to the *domain closure assumption*.

Now you should complete the following predicate definition for `meetings_one_two_three/3`. We suggest using `alldifferent/1` for the first point and defining a predicate `are_students/1` similar to `member_of/2` for the second point. Here the notation `A-B` is shorthand for defining a pair of things i.e. the predicate has 3 arguments, but each represents a pair `X-Y` that goes in there. So even though your predicate definition has a variable for each student, the query `meetings_one_two_three(Slot1, Slot2, Slot3)` will report answers like `Slot1=studentX-studentY`.[1]

```
meetings_one_two_three(A-B,C-D,E-F) :-
 % A-F are different
 % A-F are students
 % constraint 1
 % constraint 2
 % constraint 3
 % constraint 4
 % (optional, pairs are arbitrarily ordered to prevent multiple equivalent solutions)
```

If you choose to arbitrarily order students then this should place the 'larger' student (with the higher number) on the left.

The constraints above require disjunction over a single predicate e.g. `f(X)` is true for some given values of `X`. Recall that the goals in a rules are a conjunction. There are two possible approaches one could take here.

1. Define a helper-predicate that enumerates the disjuncts that are true and then simply use this predicate along with backtracking to enumerate the possible solutions. For example, if you would like to express that X=1 or X=2 at a given point, you can define a predicate `one_or_two/1` via two facts (`one_or_two(1).  one_or_two(2).`) and use `one_or_two(X)` instead.
2. Use lists to give the given values. If X=1 or X=2 then X is a member of the list [1,2].

[1] But calling `meetings_one_two_three(A,B,C,D,E,F)` with 6 arguments should not work.

Define your program in a file `meetings.pl` and once you are done submit it using the `submit` command (see the end of this document for more on submission). Note that you need to let Prolog do the work here, you should not infer the consequences of the constraints yourself and write out the solution.

Now answer the following questions in a file called `meetings.txt`

1. What is the search space of this problem e.g. the number of all possible solutions without constraints
2. How many solutions to the constraints should there be? Does your program return all possible solutions? If not comment on why not (it is not necessary to but you should understand why).
3. Does the order in which you check the constraints affect (i) the number of solutions returned, or (ii) the amount of work required to find all solutions?

Finally, think of a non-trivial constraint to add that means that there is no solution and add it to a copy of your Prolog program in `meetings_bad.pl`.

Part 2: Family Relations

Previously (in lectures) we have met the idea of modelling family relations. We'll explore that idea further here and use it to look at *recursively defined relations*. For the first few steps we will be using a provided greek god family tree (`greek_family.pl`)[2] available from `/opt/info/courses/COMP24412`.

Step 1. Non-Recursive Relations

Define the following binary relations in a file `family_relations.pl`

- `father`(X, Y). X is the father (a parent who is a man) of Y
- `mother`(X, Y). X is the mother (a parent who is a woman) of Y
- `sibling`(X, Y). X and Y have at least one parent the same but are different people.
- `brother`(X, Y). X is a man and a sibling of Y
- `sister`(X, Y). X is a woman and a sibling of Y
- `first_cousin`(X, Y). A parent of X is a sibling of the parent of Y and X and Y are different people.

You may also want to add some other relations, such as `aunt`, `niece`, `grandparent` etc.
To test these relations you can load the two files e.g.

```
1 ?- [greek_family].
2 ?- [family_relations].
```

and check a few queries do what you expect, such as

```
3 ?- father(zeus,artemis).
true
4 ?- sister(hera,X).
X = zeus ;
X = demeter.
5 ?- first_cousin(mygdon,orpheus).
true.
```

Write queries to answer the following questions:

1. Who is the father of zeus?
2. Is giles a parent of anybody?
3. Which men do not have fathers? - You can skip this one. It has been correctly pointed out that one cannot answer this query without negation and we are discouraging negation[3].
4. Which first cousins are of different gender?
5. Which men have sisters? Can we get a unique list of answers?

You should write these in a file `family_queries.txt` with one query per line (we'll read them in and run them on the appropriate files).

[2] Note that this is incomplete and also contains some relations that are not universally agreed upon. It's just a bit of fun and interesting for this example as the greek god family relations were... complex.

[3] But what if you want to answer the query? Well, firstly this can be a safe case to use negation-as-failure as described in the lecture if you ensure that the negated part of the query is fully-instantiated when it is executed. Secondly, the proper answer is that the way that we have modelled the parents relation makes this query difficult to answer and therefore we should change our representation to make it easier to answer the queries we want to ask.

Step 2. Recursive Relations

The above relations were defined in a fixed way in terms of other relations. The power of Prolog is in *recursive* relations where a relation can be defined in terms of itself. Extend `family_relations.pl` with the following two recursive relations:

1. `ancestor`(X, Y). X is an ancestor of Y if X is Y's parent or an ancestor of one of Y's parents. A person should not be their own ancestor.
2. `cousin`(X, Y). The general definition of cousin is to share a common ancestor with the common usage referring to a first cousin (you share great-grandparents with a second cousin etc). Therefore, X is a cousin of Y if X and Y share a common ancestor. A person should not be their own cousin.

Write queries (using your new relations) to answer the following questions:

1. Find all ancestors of zeus
2. Find all cousins that are not also siblings. Note that by the above definition you are cousins with your siblings. This is fine.

Add these to the file `family_queries.txt`.

Step 3. Time-Travelling God

Chronos is the greek personification of time. In mythology he created himself but let us use our imaginations and pretend he was a child of zeus and hera but also the mother and father of uranus and gaia (time is ungendered). Add these facts to `greek_familiy.pl` and check what happens when you ask for Chronos' cousins. Comment on this in a file `family.txt`.

Step 4. Reflection on Modelling

In this final step we ask you to (i) model the parent relation of a different model and (ii) reflect on the choices made in modelling and whether they could be improved.

For (i) this may be your own family, a historical family, or a fictional family. It should be non-trivial (have at least 10 people). Write this family in `my_family.pl` and make some observations about the relations in this family in `family.txt`.

For (ii) you should reflect on whether the `parents` relation is a good way of modelling a parent relation. Are there things in your family model, or in the real world, that are difficult to capture with this relation. How might you better model the parent relation? These reflections should go in `family.txt`.

Summary

For this part you should produce four files: `family_relations.pl`, `family_queries.txt`, `family.txt`, and `my_family.pl`. The file `family_queries.txt` should only contain queries (the things you type in interactive mode) or comments (starting with %). If you need any helper rules to answer your queries these should go in `family_relations.pl`. The file `family.txt` contains your answers, comments, and reflections.

Part 3: Crossing the River

Perhaps you know the following puzzle:

> A farmer is traveling with a wolf, a goat, and a cabbage to the market. On the way she encounters a river that she can only cross with a boat. Unfortunately, the boat is so small that she can take only one of the three to the other side. What makes matters worse is that when left alone, the wolf will eat the goat and the goat will eat the cabbage. Is there a way to get everyone safely to the other side? If yes, which sequence of moves will do the trick?

We will solve the puzzle in several steps: first, we will describe the datastructures and introduce some helpful predicates. Next, we will write a solution that finds the moves but will not terminate. At least we will get rid of moves that lead us back to situations we have already encountered.

Please put your rules into a file named `wolf_puzzle.pl` and document the queries you have tried in the file `wolf_queries.txt`.

Data-structures and Helper Predicates

Define a predicate `is_pos/1` that describes the sides of the river. There is no name in the puzzle, so let's call them `south` and `north`. In other words, the queries `is_pos(south).` and `is_pos(north).` will succeed but nothing else.

Define a predicate `side_opposite/2` that represents crossing the river. Therefore, `side_opposite` is true if (and only if) both arguments are opposite sides of the river. Remember that the farmer needs to be able to row back.

We also need to track the position of the farmer, the wolf, the goat and the cabbage[4]. The easiest way to groupp these up is to use a function `fwgc/4` that stores the position. Let us define a predicate `is_state` that describes what a possible state of the system is. The predicate `is_state/1` is true for terms of the shape `fwgc(Farmer,Wolf,Goat,Cabbage)` where each of the variables is one side of the river. E.g. the query `is_state(fwgc(north,south,north,north)).` will succeed but `is_state(nostate).` and `is_state(fwgc(athome,_,_,_)).` must fail.

An Inefficient Solution

Define a predicate `safestate/1` that is true when its argument is a state in which the wolf cannot eat the goat and the goat cannot eat the cabbage. Convince yourself that it always terminates by making sure `safestate(S)` reports `false`.

The next step encodes crossing the river. Define a predicate `puzzlestate_moves/2` relates the current state to a list of moves that lead there:

- Initially, everyone is in the south.
  Hint: use the empty list as starting history
- Assume we have already derived a state `S0` reachable via the moves `Ms`. Describe a new state `S` that follows the rules of the game and that is safe. Give the rule a name `M`. Then the state `S` is reachable with history `[M|Ms]`.
  Hints:
  - Make use of the helper-predicates defined above (you will not need all of them)
  - If the farmer goes alone (`M=alone`), the new state will have her on the other side but everything else stays the same.

[4] Why is there no need to track the boat?

– If the farmer takes one of the items with her (`M=wolf`, `M=goat`, `M=cabbage`), they will switch sides together and the other two items remain where they are.

When you run the query `puzzlestate_moves(fwgc(north,north,north), Moves)`, it is stuck in an infinite derivation loop. Write an explanation for why we get non-termination in a file called `wolf_explain.txt`.

You can use `isa_list(Moves), puzzlestate_moves(fwgc(north,north,north), Moves)` to enforce a fair enumeration of lists that ensures the solution is found before non-termination. Try and explain how this works in `wolf_explain.txt`.

An Efficient Solution

Based on your solution for `puzzlestate_moves/2`, define a predicate `puzzlestate_moves_without/3` that extends it with an accumulator. Similarly to the lecture, add a goal to all rules that ascertains that the new state does not repeat during the derivation. Remember that an accumulator grows during the recursion while the list of moves shrinks.

Convince yourself that `puzzlestate_moves_without(fwgc(north,north,north), Moves, [])` produces the correct solutions and write a query `puzzlestate_moves_without(State, Moves, []), false` that shows termination. Finally, write an argument for why it should always terminate in `wolf_explain.txt`.

Summary

You should submit a solution in `wolf_puzzle.pl`, a set of queries in `wolf_queries.txt`, and a file `wolf_explain.txt` with the answer to some questions posed above.

Submission and Marking

Assessment Breakdown. Recall that 70% of your marks in 24412 are for the exam, 5% are for quizzes and 25% are for coursework. There are 3 coursework assignments worth 10, 20, and 20 marks each. Therefore, one mark in this lab is worth 0.5% of your overall mark.

Time For Labs. Every 10 credit unit is meant to take you roughly 100 hours to complete. We have estimated how long you should be spending on lectures, labs, quizzes etc to convince ourselves that this course takes roughly 100 hours. In this we estimated you would spend roughly 5 hours per fortnightly exercise with 2 of these hours spent in the lab and 3 hours spent outside of the lab.

Submission. To submit you should run the *submit* command in your COMP24412/ex1 directory (where you've been writing these files). This will submit your code to an online system that you should be able to log into here

https://marking.cs.manchester.ac.uk/

If you have problems logging in then please contact Giles. From here you should be able to see your exercises and reports on the results of running tests on your submission. There will also be buttons to submit your work for marking when it is complete and to view your marks and feedback when they are available.

Marking Process. Marking in this course is *offline* e.g. it happens outside of labs. We have reserved the labs for answering questions etc. The marking will follow a *feedback-first* approach. You will receive some feedback on your submission and you will need to respond to this feedback in order to get your final mark. Research suggests that when feedback is presented alongside marks it is rarely considered in an unbiased and thorough manner. Through this approach we hope to encourage you to engage with feedback and improve the learning process.

Marking Scheme. There are 10 marks available for this exercise. In each case an excellent mark obtains 100% of marks, a good mark obtains 80% of marks, an adequate mark obtains 60% of marks and an attempt obtains 40% of marks.

- **Part 1.** *(3 marks)*
  - An *excellent* response produces a good Prolog solution that is well-written. The questions are answered fully demonstrating good understanding of the combinatorics of the problem and the execution of Prolog.
  - A *good* response produces a working Prolog solution and answers all questions adequately but not extensively.
  - An *adequate* response produces a solution that mostly works and attempts to answer the questions but there may be gaps in understanding.
- **Part 2.** *(3 marks)*
  - An *excellent* response defines all required relations and produces all suitable queries. The comments in `family.txt` demonstrate an understanding of cyclic recursive relations and give some insightful reflections on modelling.
  - A *good* response produces all required relations and queries and provides good comments but these may be lacking in deeper understanding.
  - An *adequate* response produces most of the relations and queries but some may be missing or ill-formed. Comments are present but may not fully answer questions.

- **Part 3.** *(3 marks)*
  - An *excellent* response produces a good Prolog solution that is well-written and works as expected. The efficient solution will have been given. A set of suitable queries are given. Comments show deep understanding.
  - A *good* response produces a good Prolog solution but it may not be of the efficient kind. A set of suitable queries is given. Comments show reasonable understanding.
  - An *adequate* response produces a reasonable Prolog solution and set of queries but this may not be completely correct. Comments are present but may not fully answer questions.
- **Response to Feedback.** *(1 mark)*
  - An *excellent* response addresses all points in the feedback and demonstrates a high-level of understanding.
  - A *good* response addresses some of the points in the feedback.
  - An *adequate* response comments on the feedback but in a passive way.