COMP24412
Academic Session: 2022-23

Lab 1: Prolog
Formative exercises

Joe and Giles

Note that this document contains only formative exercises. The instructions for the assessed exercises are in the other document provided. For submission information, see the assessed exercises document.

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

Additionally, if you complete the formative exercises you should be able to:

5. Explain issues surrounding termination in Prolog
6. Use an accumulator approach to detect cycles in reasoning

Part 0: Getting Started

You don't get any marks for doing the exercises in this document (see the other document provided for the assessed exercises), but the exercises here are written in more of a tutorial style, which may help you get up to speed with Prolog. In addition, solutions to many of these formative exercises will be provided on Blackboard shortly, so you can compare your solutions with them.

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.

At this point you can continue with the formative exercises, or you may prefer to switch to the assessed exercises described in the other document. Most students will benefit from following this document up to the section "Define all different".

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). Recall, the difference between unification and matching is that two terms unify if we can apply a substitution to both terms to reach two syntactically equal terms, whereas in matching you may only apply a substitution to the matching term.

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
a	=	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]`	=	a		
`[a]`	=	X		
`[X|Xs]`	=	`[A,B]`		
`[X,Y,Z]`	=	`[[A,B]|[C,D]]`		

Why does `a` not unify with `b`? It might seem obvious but we are assuming that things with unique names are unique. This is different from first-order logic, which has no such constraint. This is often referred to as the *unique names assumption*.

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. As a quick reminder `[X | Xs]` stands for a list with head `X` and tail `Xs`.

Exploring Lists

Consider the following definition for a predicate that checks whether its argument is a list.

```
isa_list([]). % The empty list is a list
isa_list([Head | Tail]) :-
isa_list(Tail). %Any head prepended to a tail list is a list"
```

This is a recursive definition. Do you understand what is going in? Try running the queries `isa_list([a,b])` and `isa_list(a)` and think about what is going on. Now use a similar approach to define a recursive predicate `member_of/2` that should succeed if the first argument is a member of its second argument (a list).

Now consider the following definition for a predicate `nonmember_of/2` that should succeed if the first argument is not a member of its second argument (a list).

```
nonmember_of(_X,[]). % Any element is not in the empty list
nonmember_of(X,[Head|Tail]) :-
dif(X,Head), % X is different from the head
nonmember_of(X, Tail). % X is not in the tail
```

The `dif` predicate is a special in-built predicate that adds the constraint that the variables X and Head are not allowed to unify to the same elements in the future - note that this is more general than the in-built `\=` predicate which requires its arguments to be sufficiently instantiated. Note also that `dif` is different from `not` as it only restricts unification rather than inverting the truth of a subgoal.

For each of the three predicates, 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

This one 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.

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) - if you're not sure what we mean by 'model' then we will revisit this later in the course.

We use terms to store structured data. For example, we can define lists using a `next` constructor instead of using the in-built syntax e.g. `next(a,next(b,empty))` for `[a,b]`. 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))` etc. If we wanted to introduce the name `two` for `succ(succ(zero))` we might try and write this as `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 the next exercise.

Scheduling Exercise

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 here.

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)
```

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

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 rule 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].

We suggest writing out our answer in a separate `meetings.pl` file. Note that you should let Prolog do the work here, you should not infer the consequences of the constraints yourself and write out the solution.

Now think about the following, preferably write down your answers to check against the solutions later:

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, can you think of a non-trivial constraint to add that means that there is no solution?

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?

This puzzle is a bit trickier and 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.

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[2]. 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.
  - 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. Why do we get non-termination here?

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. Why does this work?

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. How can we be sure that this query will always terminate?

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

Solutions and assessed exercises

Solutions to some of the exercises in this document will be provided via blackboard after a short delay to let you try them yourself.

This document contains only formative exercises: make sure you look in the other document provided to find the assessed exercises.