# Worksheet 1

# PropLog

These worksheets combine (i) notes about material covered in lectures and (ii) exercises that enable you to master the lecture material. The idea is for you to work through this worksheet, reading the tutorial material it contains and doing the exercises as you go. If you get stuck on an exercise, ask for help during a practical, during my surgery hour or at the end of a lecture. Answer sheets will be made available as the course proceeds.

In this worksheet we invite you to run and write small programs in a subset of Prolog, which we call PropLog. By doing so, you will achieve three aims:

- you will learn something of how to invoke Prolog, load programs into it and run them;
- you will learn something of what we mean when we say that proof is computation; and
- you will learn a little bit about expert systems.

## 1.1 Preliminaries

We begin with some useful information about how to run Prolog. Just read quickly through this section and then refer back to it whenever you need to.

To begin a Prolog session from the operating system (whose prompt is $) simply enter:

```
$ prolog
```

and press return.

The Prolog prompt is:

```
?-
```

Whatever you type in must end with a full stop. If it doesn't, and you hit return, then you get a different prompt:

```
|
```

You can carry on typing, but what you type is interpreted as a continuation of what you were typing before you hit return. You will only get rid of this prompt when you do put a full stop and hit return.

To exit Prolog, use the `halt` command:

```
?- halt.
```

or press CTRL-D.

CTRL-C will interrupt execution of whatever you've asked Prolog to do (e.g. if you've written something whose execution is infinite, you can stop it with CTRL-C). After pressing CTRL-C, Prolog gives you a choice of things to do: typing '`h`' will give you help, typing '`a`' will take you back to the normal Prolog prompt (`?-`), and typing '`e`' is the same as CTRL-D.

Before you invoke Prolog you will want to put some Prolog definitions into a file using an editor. (Avoid uppercase in your filename.) Then, once you're in Prolog, you need to read this file in.

Suppose you've created files called `mega1` and `mega2`. To load the file `mega1` into Prolog, you would use:

```
?- reconsult(mega1).
```

More simply and with the same effect you could type:

```
?- [-mega1].
```

(The minus sign is crucial —see footnote 1.) The latter syntax makes it easy to read in more than one file:

```
?- [-mega1, -mega2].
```

If you must use fuller pathnames then they need to be quoted, e.g.:

```
?- [-'subdir/subsubdir/mega1'].
```

If you load `mega1` into Prolog and then later, in the same session, load `mega1` again, the `reconsult` command will only read in the bits of your file(s) which have changed since you last reconsulted them in the current session, and will replace the old definitions with the new ones.[1]

Obviously when you're debugging a Prolog program you will quite often want to switch from Prolog to the editor and then back to Prolog, with the changes being reconsulted. This is not something which comes built in to Prolog. However, we have provided this facility. In fact, we've written a little Prolog program that does it. The program is in the file called `edit`, which you can copy into your own directory. Therefore, the first thing you ought to do whenever you begin a Prolog session is to load this program:

```
?- [-edit].
```

If you now type:

```
?- edit(mega1).
```

you will be put into whichever editor you have set the EDITOR environment variable to, editing the file `mega1` (or whatever file you specify). When you exit the editor, `mega1` will be reconsulted.[2]

The syntax of your program is checked as it is reconsulted.[3] Syntax errors will be signalled as they are detected. You'll obviously want to use `edit` to try to correct them. Tip: if Prolog says that there is an error in line 48 of the file, start looking in line 47!

Also useful:

```
?- shell("ls").
```

calls Unix *ls* or whatever Unix command you place between the quotes.

```
?- sh.
```

gives you a Unix shell. CTRL-D then brings you back to Prolog.

## 1.2 PropLog

PropLog is a subset of Prolog. You are going to write programs in it for the purposes of this worksheet.

To use PropLog:

[1]There is a command, `consult`, which will *duplicate* definitions if you use it to load a file more than once. (This is also the effect you get if you leave the minus sign out when loading files.) Avoid this command!

[2]The alternative to using the `edit` program is to have your editor running in one window and Prolog running in another, but then, when you finish editing you must remember *firstly to save the changes and secondly to reconsult the file.*

[3]There is no *compiler*. Prolog is an *interpreted* language. (Actually, you *can* get compilers, but we won't use one in this course.)

1. Put a PropLog program into a file using an editor, avoiding uppercase in your filename. (We explain what PropLog programs look like below.)
2. Invoke Prolog (by typing `prolog` at the Unix prompt).
3. Reconsult (i.e. load) the `edit` program by typing:

   ```
   ?- [-edit].
   ```

4. Reconsult your PropLog program (e.g. if it is called `mega1`, you enter `[-mega1]`).

You will then be ready to run your program.

A PropLog program is a collection of *clauses*. These clauses may be either *facts* or *rules*. We will write a PropLog program that uses both facts and rules to help us to identify animals we encounter on a jungle safari.

### 1.2.1 Simple facts and queries

Here is a simple PropLog program/database, which you might enter into a file.

```
/* A Zoological Example                               */

zebedee_isa_mammal.
zebedee_has_hooves.
zebedee_has_stripes.
```

This program/database contains one comment line. Comment lines are either enclosed between the symbols `/*` and `*/`, in which case they may stretch over several lines, or they may simply be preceded by `%`, in which case they extend only as far as the end of the line. The program/database then contains three clauses, each of which, in this case, is a fact. Each fact consists simply of what is called a predicate symbol; these can be (pretty much) any sequence of characters as long as they begin with a lowercase letter. Each fact must end with a full stop.

□ Put these facts into a file using an editor, start Prolog, and read in `edit` and your file of facts.

Once these facts have been loaded in, you can ask queries by entering *goals*.

□ Confirm that you get the same answers as below and in the subsequent queries in this subsection:

```
?- zebedee_isa_mammal.
yes
```

```
?- zebedee_isa_zebra.
no
```

Subtle point (which you might not understand at this stage):

> *no* is not the same as false; it means the goal is not provable.
> *yes* is not the same as true; it means the goal is provable.

You can conjoin goals. In the following query we check whether Zebedee is a mammal and has hooves:

```
?- zebedee_isa_mammal, zebedee_has_hooves.
yes
```

The comma between the two subgoals is interpreted as 'and' ($\wedge$). Since the goals are conjoined using $\wedge$, we only get the answer *yes* if *both* goals are proved. If one or both are not proved, then the answer will be *no*:

```
?- zebedee_isa_mammal, zebedee_has_spots.
no
```

Given what you should know about $\wedge$ in logic, it should come as no surprise that the queries give the same answers as above when the order of the conjoined goals is reversed:

```
?- zebedee_has_hooves, zebedee_isa_mammal.
yes

?- zebedee_has_spots, zebedee_isa_mammal.
no
```

### 1.2.2 Simple rules

Our next type of clause is the *rule*. Rules enable us to define new relationships in terms of existing relationships. For example, here's a definition which enables us to determine whether Zebedee is an ungulate:

```
zebedee_isa_ungulate :- zebedee_isa_mammal, zebedee_has_hooves.
```

To the left of the symbol `:-` is the *head* of the rule, to the right is the *body*. You can think of the head as naming the relationship or goal, and the body as listing subgoals which must be proved in order to satisfy the goal. (There may be as many subgoals as you like.) Equally, you can see that this is like giving a name to the conjoined query that we issued earlier, thus making it easier to issue that query repeatedly but with less typing.

The symbol `:-` is read as *"if"*. So this rule can be read as: `zebedee_isa_ungulate` if `zebedee_isa_mammal` and `zebedee_has_hooves`. The *"if"* used here is the 'if' of logic, i.e. the rule is an alternative notation for

(`zebedee_isa_mammal` $\wedge$ `zebedee_has_hooves`) $\rightarrow$ `zebedee_isa_ungulate`

Once this rule has been loaded into memory, you can query it in the same way as you would query the more basic predicates.

□ Use the editor to add the rule to the end of your file, reconsult the file, and then try this query out.

```
?- zebedee_isa_ungulate.
yes
```

Faced with this query, the system tries to find a match somewhere in its database. It finds our rule because the query matches the head of the rule. Now, to find out whether to return *yes* or *no*, the system poses itself the body of the rule as subgoals that it must prove. In other words, the original goal of proving `animal_isa_ungulate` is expanded into the subgoals of proving that `zebedee_isa_mammal` *and* `zebedee_has_hooves`. Since both of these *are* true, the answer is *yes*.

Whole chains of this sort of behaviour can be set up.

□ Add the following rule to your file and bring it into memory:

```
zebedee_isa_zebra :- zebedee_isa_ungulate, zebedee_has_stripes.
```

Then query the new predicate:

```
?- zebedee_isa_zebra.
yes
```

Here the original goal, `zebedee_isa_zebra`, is expanded into proving that `zebedee_isa_ungulate` and `zebedee_has_stripes`. The second of these is true. The first is not directly true, but can be expanded using our rule about ungulates into showing that `zebedee_isa_mammal` and `zebedee_has_hooves`, both of which *are* true so the answer is *yes*.

You have seen that rules behave pretty much like conjunctive queries. Suppose now that we want to define a new predicate, `zebedee_has_camouflage`. In our simple view of zoology, Zebedee has camouflage if he has stripes *or* if he has spots. How do we capture the *"or"* here?

The answer is that we use several *separate* clauses all defining the same predicate, as follows:

```
zebedee_has_camouflage :- zebedee_has_stripes.
zebedee_has_camouflage :- zebedee_has_spots.
```

We have one clause for each case of the definition.

□ Add the definition of `zebedee_has_camouflage` to your file and try it out:

```
?- zebedee_has_camouflage.
yes
```

The behaviour of the system faced with this query is again to try to find a match. But this time there are multiple matches, i.e. two clauses match the goal. It will return ***yes*** if the body of *at least one* of them can be proved. In the example above, the first rule will succeed, although the second won't. Since there has been one success, ***yes*** is returned.

**Exercise 1.1** *Expand your system's knowledge of zoology by adding rules such as those that correspond to the English below:*

- *Zebedee is a rhino if he has a horn and he has leathery skin.*
- *Zebedee is a carnivore if he is a mammal and eats meat, or if he has claws, pointed teeth, and his eyes point forward.*
- *If Zebedee has feathers, he is a bird. Alternatively, if he flies and lays eggs, he is a bird.*
- *If Zebedee is a carnivore and has spots, he is a cheetah.*
- *We can tell whether Zebedee is a tiger because we know that a tiger is a carnivore that has stripes.*
- *Similarly, we know that giraffes are ungulates with spots and a long neck.*

*When you have done this, you can issue more interesting goals. Of course, given that we have only three facts in the database (Zebedee is a mammal and has hooves and stripes) most answers will be* `no`. *Experiment by changing the facts, e.g. delete the information about stripes and add in information about spots and a long neck in order to test your rule for giraffes.*

**Exercise 1.2** *Satisfy yourself that the following statements are true by issuing some queries, noting their answers and then shuffling the order of things around and re-running the queries.*

- *The order of conjuncts in a query is not significant in PropLog.*
- *The order of conjuncts in the body of a rule is not significant in PropLog.*
- *The order of clauses is not significant in PropLog.*

*N.B. The above facts are true of PropLog but are* not *true of Prolog in general.*

**Exercise 1.3** *Just for interest, add the following rule:*

```
zebedee_isa_elephant :- zebedee_isa_elephant.
```

*Run the following query:*

```
?- zebedee_isa_elephant.
```

*. . . and see what happens.*

## 1.3 Concluding Remarks

1. You ought to be able to spot that there is some sort of inference process/theorem proving going on in PropLog. You have encountered the inference rule of Modus Ponens before now, which states that if you have a formula of form $P \rightarrow Q$ for some propositions $P$ and $Q$, and a formula of form $P$, then you can derive $Q$. In our case, analogously, we have rules which, in the simplest case, are of the form $Q$ `:-` $P$, and facts of the form $P$, and this allows us to return *yes* when asked to prove $Q$. (In fact, something a bit more general is going on.) Thus you can see that we have a theorem-prover for some propositional logic. But what's more, we can see that proof can be a form of computation.

2. The jungle zoology example is really a very simple example of a type of AI program called an *expert system*. These are systems that are supposed to capture human expertise for carrying out specialised tasks. Expert knowledge about animal identification in this case (but it could equally well have been about heart disease or stock market investments) has been encoded as rules that chain together to lead to conclusions. This sort of idea has formed the basis for the implementation of a large subclass of expert systems.

**Acknowledgements**

The inspiration for the jungle zoology expert system came from P.H.Winston: *Artificial Intelligence*, 2nd edn., Addison-Wesley, 1984, pp.177-180. The inspiration for beginning the course with a language based on propositional logic came from D.Maier and D.S.Warren: *Computing with Logic*, Benjamin/Cummings, 1988. The `edit` program was written by Colin Runciman.