# Worksheet 2

# Introduction to DataLog

In this worksheet we introduce DataLog, which is a generalisation of PropLog but still a subset of Prolog. The aims of the worksheet are that:

- you learn something of how some internal structure to the knowledge we want to represent can be captured by the use of predicate symbols and their arguments;
- you see some simple database-style applications of DataLog;
- you are introduced to the notion of polymodality.

## 2.1 Preliminaries

You run DataLog in exactly the same way as you run PropLog, *viz.:*

- Put a DataLog program into a file using an editor.
- Invoke Prolog by typing `prolog` at the Unix prompt.
- Reconsult into Prolog the `edit` program and your own program.
- Issue goals as before.

The first thing to note is that everything you can do in PropLog can also be done in DataLog: DataLog is a generalisation of PropLog. A major limitation of PropLog was that all the statements were 'particular' to an individual (Zebedee in our examples). We really want to be able to parameterise them, i.e. we want to be able to write rules and facts that apply to *any* individual. To do this, we need to be able to express statements with some substructure to them. We achieve this by the introduction of predicate symbols and their arguments, these arguments, at this stage, being restricted to constant symbols and variables.

## 2.2 Simple facts and queries

Here is a simple DataLog program/database, based on, but more complicated than, the Zoology example from worksheet 1, which is in the file `zoo`; you can copy this file into your own directory and use it in tackling this worksheet.

```
/* A Zoological Example                            */

isa_mammal(gertie).
isa_mammal(zebedee).
isa_mammal(tigger).

has_feathers(tweety).
has_hooves(gertie).
has_hooves(zebedee).
has_spots(gertie).
has_long_neck(gertie).
has_stripes(zebedee).
has_pointed_teeth(tigger).
has_claws(tigger).
has_forward_eyes(tigger).

colour(tweety, blue).
colour(gertie, tawny).
colour(zebedee, black_and_white).
colour(tigger, tawny).
```

These facts are 'parameterised'.[1] Each consists of a *predicate symbol*, such as `isa_mammal`, `has_feathers`, `has_hooves`, `has_spots`, `has_long_neck`, `has_stripes`, `has_pointed_teeth`, `has_claws`, `has_forward_eyes` and `colour`, and constant symbols, such as `gertie`, `zebedee`, `tigger`, `tweety`, `blue`, `tawny` and `black_and_white`, which we are using as the arguments to the predicates. (Note that there is no space between the predicate symbol and the open parenthesis that follows it.) This parameterisation has made it sensible to include facts about several different creatures in the same program. We will later write rules that express generalisations about these individual animals.

Predicate symbols and constant symbols can be pretty much any string of letters beginning with a lowercase letter. (In fact, constant symbols can also be numbers, certain sequences of special characters, or any string of characters enclosed in single quotes, e.g. `'Rupert Bear'`.)

DataLog goals can similarly be parameterised.

[1] This is a bit like setting up 10 relations in a relational database as in the ISD course, except that attributes (column headings) are not given and we aren't drawing them as a table.

□ Put the facts into a file, load them in and confirm that you get the same answers as below and in the subsequent queries in this subsection:

```
?- isa_mammal(zebedee).
yes

?- colour(tweety, green).
no
```

Queries can also contain *variables*. Variables in DataLog (and in Prolog in general) stand for an unspecified entity, rather than name a storage location in memory. They are therefore not like variables in Ada, which name storage locations. A reasonable analogy is wild cards in *grep* or similar pattern matchers. When a goal has a variable in it, DataLog must find a suitable value to substitute for that variable in such a way as to make the goal true.

DataLog will display the first value that it can find which satisfies the goal and then 'hangs'. You enter a semi-colon and press return to get the next solution. You can obviously keep typing semi-colon-return until DataLog gives a *no*, which means that it can find no more solutions. On the other hand, if you don't want DataLog to search for further solutions, then simply type return (i.e. don't type the semi-colon).

```
?- isa_mammal(X).
X = gertie ;
X = zebedee ;
X = tigger ;
no
```

Note variables always begin with capital letters (unlike normal maths notation):

```
?- colour(zebedee, What).
What = black_and_white ;
no

?- colour(Animal, Colour).
Colour = blue
Animal = tweety ;

Colour = tawny
Animal = gertie ;

Colour = black_and_white
Animal = zebedee ;

Colour = tawny
```

```
Animal = tigger ;

no
```

You can conjoin goals as before.

```
?-has_hooves(gertie), isa_mammal(gertie).
yes
```

This gets more interesting when the two subgoals have shared variables. Here we ask for hooved mammals, i.e. something, `X`, which has hooves and which is also a mammal:

```
?- has_hooves(X), isa_mammal(X).
X = gertie ;
X = zebedee ;
no
```

This query insists that whatever value can be found to substitute for `X` in the first subgoal must also be substituted for `X` in the second.

**Exercise 2.1** *Write a query that returns the names of animals that have spots and a long neck.*

Important point: you cannot put a variable where a predicate symbol should be:

```
?- X(tweety).
Syntax error
```

This will give an error; it will *not* give `X = has_feathers`. This is because DataLog (and Prolog) are based on *first-order* logic; we explain this elsewhere in the course.

## 2.3 Simple rules

Rules in DataLog are like rules in PropLog: they define new relationships in terms of existing relationships. You can also think of them as named queries.[2] But since they can now contain variables they are much more useful. For example, here are the rules that define ungulates, zebras and camouflaged beasts:

[2]Since rules are like named queries, they are also like views in relational databases (as in the ISD course), but they are more powerful, as we shall see.

```
isa_ungulate(X) :- isa_mammal(X), has_hooves(X).

isa_zebra(Y) :- isa_ungulate(Y), has_stripes(Y).

has_camouflage(Z1) :- has_stripes(Z1).
has_camouflage(Z2) :- has_spots(Z2).
```

This is much more general than our version in the previous worksheet which worked only for Zebedee. The variables in rules are universally quantified. The first rule, for example, can be paraphrased into English as: 'for all `X`, if `X` is a mammal and `X` has hooves, then `X` is an ungulate'.

□ Use the editor to add the rules to the end of your file, and then try these queries out.

```
?- isa_ungulate(zebedee).
yes

?- isa_ungulate(X).
X = gertie ;

X = zebedee ;

no

?- isa_zebra(Animal).
Animal = zebedee ;

no
```

As you can see in the last example, you don't have to use the same variable name as you used to define the rule. The variable `Animal` is substituted for the variable `X` in your definition.

In the above rules, I used different variables in each rule. This was *not* necessary. Variables are *local* to clauses. If a variable, `X` say, appears more than once in one particular clause (e.g. in several places in a rule or conjunctive query), then those occurrences notify the *same* variable. However, if there is an occurrence of `X` in one clause and an occurrence in another clause, these do *not* notify the same variable. Hence, the following would have been just as good:

```
isa_ungulate(X) :- isa_mammal(X), has_hooves(X).

isa_zebra(X) :- isa_ungulate(X), has_stripes(X).

has_camouflage(X) :- has_stripes(X).
has_camouflage(X) :- has_spots(X).
```

The way DataLog manages not to get confused is that, whenever it picks up a clause, it renames all the variables first before doing any other processing. You can therefore see that it doesn't really matter what you call your variables as long as (1) they begin with an uppercase letter or an underscore, and (2) within a particular clause, you use the same name when you want something to have the same binding as something else in that clause.

There is an exception to this: the so-called *anonymous variable*. Anonymous variables are used in 'don't care' situations (where you don't care what matches with the variable and you do not want values that do match with it to be displayed). The anonymous variable is written as an underscore.

For example, the following query has two variables. The first is anonymous and will match with anything and the thing it matches with is not displayed:

```
?- colour(_, C).
C = blue ;
C = tawny ;
etc.
```

Similarly, in the following rule, we compute the names of the coloured mammals:

```
isa_coloured_mammal(X) :- isa_mammal(X), colour(X, _).
```

which says that the actual colour of the mammal is not important.

Multiple occurrences of the anonymous variable within the same clause do not have to match. For example, the following two uses of the anonymous variable do not have to be instantiated to the same value.

```
?- colour(_, _).
yes
```

If they did have to match, the answer would clearly be *no* (because our database contains no animals whose name is the same as their colour), just as the answer to the query `?-colour(X, X)` would be *no*. But since they do not have to be instantiated to the same value, but can both be instantiated to any value, they successfully match facts in the database and hence return the answer *yes*. (They don't return actual pairs of values because values bound to anonymous variables are not displayed.)

**Exercise 2.2** *Rewrite your answers to exercise 1.1 from worksheet 1 so as to generalise the definitions of* `rhino`, `carnivore`, `bird`, `cheetah`, `tiger` *and* `giraffe`. *Try them out. (You will want to add more facts too.)*

**Exercise 2.3** *Henri owns a restaurant which has the following dishes available. (These facts are stored in the file* `chez_henri`, *which you can copy into your own directory.)*

```
hors_d_oeuvre(artichauts_melanie).
hors_d_oeuvre(truffes_sous_le_sel).
hors_d_oeuvre(cresson_oeuf_poche).

meat(grillade_de_boeuf).
meat(poulet_au_tileul).

fish(bar_aux_algues).
fish(chapon_farci).

dessert(sorbet_aux_poires).
dessert(fraises_chantilly).
dessert(melon_en_surprise).
```

*a) Define the predicate (i.e. give a rule for)* `maincourse(M)`*, where* `M` *is a* `maincourse` *if it is either a meat dish or a fish dish.*

*b) Define the predicate* `meal(H, M, D)`*, where a meal comprises an hors d'oeuvre,* `H`*, a main course,* `M` *and a dessert,* `D`*.*

*c) Write a conjunctive* query *to find which meals have fish as their main course. (Note: I want a* query *here, not a* rule*.)*

*d) Write a* query *to find out which meals contain artichauts melanie as hors d'oeuvre.*

*e) Complete the program so that a* `full_meal` *contains a drink, which can be either wine, beer or mineral water.*

**Exercise 2.4** *This Prolog program concerns the* Achin' Hearts Dating Agency*.*

*We have entered some facts into the file* `dating`*, which you can copy into your own directory, about men registered with the agency. The format of the facts is* `man(Name, Size, Hair_colour, Age)`*, where* `Size` *can be* `large`, `medium` *or* `small`*;* `Hair_colour` *can be* `blond`, `brown`, `red` *or* `light_brown`*;* `Age` *can be* `young`, `mature` *or* `old`*.*

*We've also entered similar facts about women registered with the agency, using the format* `woman(Name, Size, Hair_colour, Age)`*.*

*Each person has their tastes recorded using predicates of the form* `tastes(Name, Music, Literature, Sport)`*, where* `Music` *can be* `classical`, `pop` *or* `jazz`*;* `Literature` *can be* `adventure`, `sci_fi` *or* `detective`*;* `Sport` *can be* `tennis`, `swimming` *or* `jogging`*.*

*And for each person we have recorded the physical characteristics of the person they're looking for using the* `seeks` *predicate:* `seeks(Name, Size, Hair_colour, Age)` *means that* `Name` *seeks someone with the physical characteristics listed.*

*a) Note that some of the facts in the file contain variables. What do you think that facts with variables in them 'mean'?*

*b)* `X` *suits* `Y` *physically if* `X` *is a man,* `Y` *is a woman, and the size, hair colour and age of* `Y` *are what* `X` *seeks; and, the size, hair colour and age of* `X` *are what* `Y` *seeks. Define a rule for* `suits_physically(X, Y)`.

*c)* `X` *and* `Y` *have the same tastes if their recorded tastes in music, literature and sport coincide exactly. Define a rule for* `have_same_tastes(X, Y)`. *(You don't have to check that their gender is different; after all, two people of the same gender can have the same tastes. This removes one problem for you. However, your definition will probably show a person having the same tastes as him/herself. You should think about solutions to this problem, but you need not try to overcome it.)*

*d)* `X` *and* `Y` *are matched if they are physically suited and have the same tastes. Define a rule for* `matched(X, Y)`.

*Try out your rules on the facts in the* `dating` *file.*

## 2.4 Polymodality

Earlier we gave a definition of `isa_ungulate`, repeated here for convenience:

```
isa_ungulate(X) :- isa_mammal(X), has_hooves(X).
```

This definition (and other definitions) is very flexible. It can be used in one of two *modes*.

In the goal/query:

```
?- isa_ungulate(gertie).
```

the variable `X` in the definition is *instantiated*: it is mapped to a constant symbol (`gertie`). (Similarly, in the goal

```
?- colour(Y, tawny), isa_ungulate(Y).
```

the variable `X` is again instantiated: by the time the call to `isa_ungulate` is made, `Y` will have been bound to the name of a tawny animal, this being a constant symbol, and this is what `X` will be mapped to.) A variable is instantiated if it is mapped to a constant symbol.[3]

In the goal/query:

```
?- isa_ungulate(Y).
```

the variable `X` is *uninstantiated*: it is mapped to a variable (`Y`). (Similarly, in the goal

[3] Be warned that I will revise this definition later; this is only a first approximation which would not get full credit if given as an exam answer.

```
?- isa_ungulate(X).
```

the variable `X` in the definition of `isa_ungulate` is uninstantiated as it is mapped to itself, which is a variable, of course.) A variable is uninstantiated if it is mapped to itself or another variable.[4]

Where `X` was instantiated, Prolog attempts to prove that the goal follows from the program; where it is uninstantiated, Prolog attempts to find bindings of values to the variable that give statements that follow from the program. These are the two modes of the predicate `isa_ungulate`.

Consider the following more radical example.

In most programming languages we would have to write separate procedures or function definitions for each of the following operations:

1. given a beast, compute another beast that would be a tasty snack for the given beast;
2. given a beast, compute another beast for whom the given beast would be a tasty snack;
3. given two beasts, check whether the second would be a tasty snack for the first; and
4. compute all pairs of beasts where the second is a tasty snack for the first.

The same relation is involved in all the above cases. This is reflected by the fact that in Prolog we need give only one definition, e.g.:

```
tasty_snack(Beast1, Beast2) :- carnivore(Beast1), isa_ungulate(Beast2).
```

To get the effect of the four operations, we use different queries which call the definition with different modes by varying the pattern of instantiated and uninstantiated arguments, e.g.:

| | | |
|---|---|---|
| 1. | First instantiated; second uninstantiated: | `?- tasty_snack(tigger, X).` |
| 2. | First uninstantiated; second instantiated: | `?- tasty_snack(X, zebedee).` |
| 3. | Both instantiated: | `?- tasty_snack(tweety, tigger).` |
| 4. | Both uninstantiated: | `?- tasty_snack(X, Y).` |

There is no notion of certain parameters being fixed as the 'input parameters' and others being fixed as the 'output parameters'.

This feature of Prolog, of logic programming languages in general, and of relational programming languages in general is referred to as *polymodality* ('poly'

[4]The same warning as in the previous footnote applies here too.

meaning 'many', hence 'many-modes'). It is not a feature of functional languages,[5] nor is it a built-in feature of imperative languages.[6]

Polymodality is undoubtedly a useful feature. We shall see many uses of it. For now, consider the following:

- Polymodal addition: given a definition of `add(X, Y, Z)`, which is true if `Z` is the sum of `X` and `Y`, we get the following modes (among others):
  - instantiate `X` and `Y` to numbers, bind `Z` to their sum;
  - instantiate `Y` and `Z` to numbers, bind `X` to the difference between `Z` and `Y` (hence giving the effect of subtraction);
- Polymodal list concatenation: given a definition of `append(Xs, Ys, Zs)`, which is true if `Zs` is the list comprising the list `Ys` stuck on to the end of the list `Xs`, we get the following modes (among others):
  - instantiate `Xs` and `Ys` to lists, bind `Zs` to the concatenation of `Xs` and `Ys`;
  - instantiate `Xs`, `Ys` and `Zs`, check that the list bound to `Zs` is the concatenation of the lists bound to `Xs` and `Ys`;
  - instantiate `Zs` to a list, bind `Xs` and `Ys` to all its prefix/suffix pairs
  - instantiate `Xs` and `Zs` to lists, bind `Ys` to the corresponding suffix;
  - instantiate `Ys` and `Zs` to lists, bind `Xs` to the corresponding prefix.

The benefits of polymodality include:

- Programs can be more concise than when there are multiple definitions for different modes;
- Relationships will be expressed in one place only, so there is less danger of inconsistency than there would be with different definitions for different modes.

On the downside, it will not always be possible to give simple definitions that are polymodal:

- the best order of subgoals in a rule that is needed for efficiency may conflict with the order needed for polymodality;
- some of the aspects of Prolog that give us 'impure Prolog' are incompatible with polymodality.

[5]Do not confuse it with the *polymorphism* of functional languages where the type of a function definition is allowed to include type variables.

[6]Of course, with some effort, you *can* achieve something of similar effect: you might provide multiple pieces of code (one per mode) and some form of switch/case statement that invokes the appropriate piece of code depending on the arguments being supplied. The point about this in Prolog though is that you need only one definition; there's no switching between different definitions.

**Acknowledgement**

- Datalog is a language that is being used in many research centres as a research vehicle for investigating the integration of logic programming and database technology. The idea to use it as a teaching vehicle came from the following two books:
  - F.C.N.Pereira and S.M.Shieber: *Prolog and Natural-Language Analysis*, CSLI. 1987
  - D.Maier and D.S.Warren: *Computing with Logic*, Benjamin/Cummings, 1988
- Some of the exercises in this worksheet are based on exercises to be found in
  - F.Giannesini et al.: *Prolog*, Addison-Wesley, 1986

  (Note: their dialect of Prolog is very different from ours.)