Worksheet 3

Recursion in DataLog

In this worksheet we introduce recursive rules as a way of gaining even greater power. The aim of the worksheet is that

- you will understand and be able to write recursive DataLog rules.

3.1 Recursive rules

Rules can be *recursive*. Roughly speaking, a rule is recursive if the predicate is defined in terms of itself (perhaps more strictly, if the predicate is defined in terms of itself applied to simpler arguments).

A very simple example is as follows:

```
likes(gertie, zebedee).
likes(tigger, zebedee).

likes(zebedee, X) :- likes(X, zebedee).
```

The last of these three clauses is recursive and states that Zebedee likes those animals that like him. Consider the following query:

```
?- likes(zebedee, gertie).
yes
```

DataLog tries to match the query with one of the clauses. It matches only with the head of the last clause, with `gertie` being substituted for `X`. This thus sets up the subgoal of solving the query `likes(gertie, zebedee)` (i.e. the body of the rule after substitution). This matches the first clause, thus giving us the answer *yes*.

If we use the following goal:

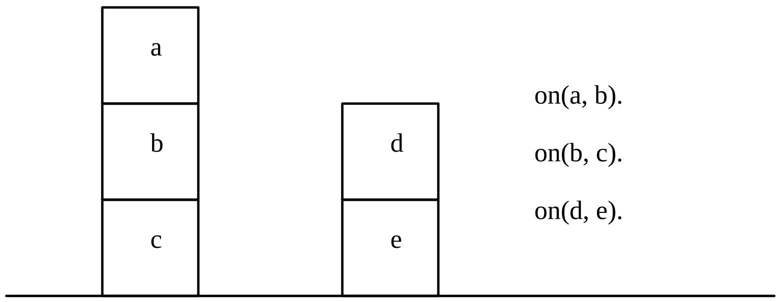
```
?-likes(zebedee, Who).
Who = gertie ;
Who = tigger ;
```

after giving the two answers, Prolog runs out of space. This is because the goal launches an infinite computation. This is something we have to be very careful about when writing recursive definitions.

Only when both arguments are instantiated does the program terminate with an answer. For all other patterns of instantiation, the definition launches an infinite computation. This definition is not polymodal; it offers only one mode of operation. We will need to reconsider polymodality later in this worksheet.

A slightly more complicated example follows.

Suppose we have some blocks stacked on top of each other on a table. We will represent this using a predicate `on`, where `on(Block1, Block2)` is true if `Block1` is (directly) on `Block2`, i.e. they are touching, with `Block1` on top (as in the figure).



We want to define a predicate `above(Block1, Block2)` that is true if `Block1` is (somewhere) above `Block2` in the stack. (As it happens, `above` is the transitive closure of `on`.)

We could start by writing clauses such as:

```
above(B1, B2) :- on(B1, B2).
above(B1, B2) :- on(B1, Middle), on(Middle, B2).
above(B1, B2) :- on(B1, Middle1), on(Middle1, Middle2), on(Middle2, B2).

etc.
```

There are enough clauses here if we know that our stacks of blocks are never more than three blocks high. But if they can be four high, we need another clause. And, if they can be five high, we need yet another. And so on. Clearly, no finite statement of the relationship for an arbitrary number of blocks is possible using this approach.

Instead, we define `above` recursively. In the simplest case, one block, `B1`, is above another, `B2`, if `B1` is directly on top of `B2`:

```
above(B1, B2) :- on(B1, B2).
```

And, in the more general case, a block, `B1` is above another, `B2`, if `B1` is on top of another block, `Middle`, which itself is somewhere above `B2`:

```
above(B1, B2) :- on(B1, Middle), above(Middle, B2).
```

These two rules (clauses) *together* define `above`. The first rule is what is called the *base case* or *boundary case*, and the second is the *step case* or *recursive case*.

Obviously, it is the second clause, the recursive one, that is giving us the power. But, without the first clause, we would launch an infinite computation. It is the second clause that causes particular computations to 'bottom out'.

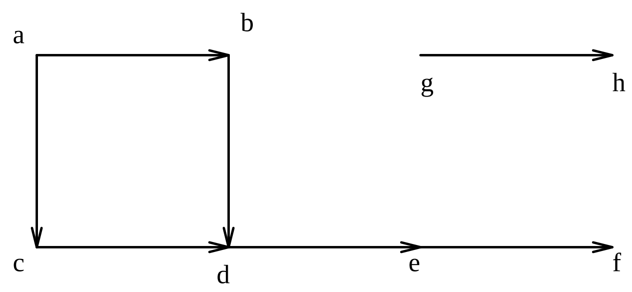
Think about how the following goal would be proved:

```
?- above(a, c).
yes
```

Of the two clauses that define `above`, if the system matches this goal with the first clause, it gives itself the job of proving that `on(a, c)` holds. This fails. But if it uses the match with the second clause, it gives itself the job of proving that `on(a, Middle)` and `above(Middle, c)` hold. The first of these is easily proved by matching with one of the `on` facts, and instantiating `Middle` to `b`. This turns the other goal into the job of proving `above(b, c)`. This is the recursive call to the `above` predicate, and launches a similar computation to that of the original goal. Specifically, again, it can try matches with either of the two clauses defining `above`. If it matches with the first clause, the system is given the job of proving `on(b, c)`, which succeeds (it matches one of the facts), and so the answer *yes* can be returned.

Here's another example.

A simple directed graph is shown in the figure:



Represented as DataLog clauses we get:

```
edge(a, b).
edge(a, c).
edge(b, d).
edge(c, d).
edge(d, e).
edge(e, f).
edge(g, h).
```

Two nodes are connected if there is a series of edges that lead from the first node to the second. The simple base case is where the nodes are directly connected by a single edge. The step case says that `Node1` is connected to `Node2` if there is an edge from `Node1` to some intermediary node, `Int`, and then `Int` is connected to `Node2`:

```
connected(Node1, Node2) :- edge(Node1, Node2).
connected(Node1, Node2) :- edge(Node1, Int), connected(Int, Node2).
```

□ Enter the facts and rules for the boxes example and the graph example and try them out. (Note that this simple recursive definition gives multiple answers when there are multiple paths.)

3.2 Polymodality revisited

The definition we have given of `above`, redisplayed here and renamed to `above1` for convenience:

```
above1(B1, B2) :- on(B1, B2).
above1(B1, B2) :- on(B1, Middle), above1(Middle, B2).
```

is fully polymodal. It gives a terminating computation for all possible modes (i.e. for all four possible instantiation patterns of its arguments).

□ Confirm the previous statement by querying `above1` using a variety of goals with different instantiation patterns.

However, the following definition:

```
above2(B1, B2) :- above2(Middle, B2), on(B1, Middle).
above2(B1, B2) :- on(B1, B2).
```

launches an infinite computation irrespective of the mode used in the query.

□ Confirm this.

The following definition:

```
above3(B1, B2) :- on(B1, B2).
above3(B1, B2) :- above3(Middle, B2), on(B1, Middle).
```

and the next definition, the likes of which I have seen some students try in the past:

```
above4(B1, B2) :- on(B1, B2).
above4(B1, B2) :- above4(B1, Middle), above5(Middle, B2).
```

are partly polymodal. They allow certain patterns of instantiation but not others (others launch infinite computations).

□ Of the four possible patterns of instantiation, for which ones are `above3` and `above4` properly defined and for which do they launch infinite computations? (Note that they will produce answers prior to running out of stack space, which is somewhat better than the definition of `above2`, which produced nothing of any use at all.)

Finally, the following definition

```
above5(B1, B2) :- on(B1, Middle), above5(Middle, B2).
above5(B1, B2) :- on(B1, B2).
```

is defined for all modes, just like `above1`.

□ Confirm this.

What is going on here? If you look at the definitions that are not fully polymodal, you will see that they are the ones in which a recursive call comes leftmost in the body of the recursive clause. In `above1` and `above5`, where things are fine, the recursive call does not come leftmost. This leads me to the following rule of thumb:

RULE OF THUMB

The order of the subgoals in a rule is VERY IMPORTANT in Prolog: as a rule of thumb do not have the recursive call as the first subgoal in the rule's body.

Of course, this doesn't explain *why* the order of the subgoals is important. We were treating the commas between subgoals as if they were the same as logical conjunction ($\wedge$), and we know that $P \wedge Q \equiv Q \wedge P$, so the order shouldn't matter. We also did some experiments with different orders in worksheet 1, and found the order didn't matter there either. It is only now that we have introduced recursion that the order becomes important. The full explanation will be given much later in the course, when we consider the declarative and procedural semantics of Prolog. For now, we note that in recursive definitions, the order is important, and we have a rule of thumb for dealing with it.

In fact, we have to abandon this simple rule of thumb in worksheet 5, where sometimes we will be forced into putting the recursive call leftmost; in those cases, we will provide definitions similar to `above3`, which do at least work in some modes; clearly, definitions similar to `above2`, which do not work in any mode, are no good at all. So this leads to another rule of thumb, one that helps us to prefer definitions such as `above3` to those such as `above2`; the difference between these two definitions is in the ordering of the clauses:

RULE OF THUMB

The order of the clauses in a definition is VERY IMPORTANT in Prolog: as a rule of thumb put the base case as the first clause and the recursive case(s) as subsequent clauses.

Use the rules of thumb when answering the following exercises.

Exercise 3.1 *Create a file of parentage facts about your family, or copy the file* `family`*, which contains the following fictitious family, into your directory:*

```
parent(vicky, ed).
parent(albert, ed).
parent(ed, georgie).
parent(alexandra, georgie).
parent(georgie, george).
parent(mary, george).
parent(george, liz).
parent(elizabeth, liz).
parent(liz, chas).
parent(phil, chas).
parent(chas, willie).
parent(chas, henry).
parent(di, willie).
parent(di, henry).
```

a) Give (non-recursive) definitions of `grandparent` *and* `great_grandparent`*. Check that they are fully polymodal.*

b) Give a (recursive) definition for `ancestor(A, D)`*, such that* `A` *is the ancestor of* `D` *if they are linked by an arbitrarily long parentage chain. Check that your definition is fully polymodal. (If it is not, re-read the rules of thumb!)*

Exercise 3.2 *Copy the file* `evolution`*, which contains some facts about what evolved from what, such as:*

```
evolved(homo_sapiens, neanderthal).
evolved(neanderthal, homo_erectus).
etc.
```

The predicate `higher(C1, C2)` *is satisfied if* `C1` *is anywhere higher in the evolutionary chain than* `C2`*. (By "higher" we mean that* `C1` *is more advanced (less primitive) than* `C2`*.) For example:*

```
?- higher(homo_sapiens, X).

X = neanderthal ;

X = homo_erectus ;
etc.
```

Define `higher(C1, C2)`*. (Make sure that it is fully polymodal.)*

3.3 Debugging facilities

Once one starts to write recursive programs, there is the potential to get things wrong. So now seems like a good point to show some of the Prolog interpreter's debugging facilities.

To use these facilities, we need to have a way of telling Prolog which rules we are talking about. We do this using the predicate's *name* and, in some cases, its *arity*. The *arity* is the number of arguments which a predicate takes. In Prolog it is possible to define two predicates which have the same name but different arities, e.g.:

```
pred(X, Y) :- etc.
pred(X) :- etc.
```

Here there are two *completely different* predicates. Both are called `pred` but their arities differ. There is normally no confusion (at least as far as the machine is concerned) because when you enter a query, the number of arguments you give in your query will enable Prolog to determine which version of `pred` you are referring to.

However, when this isn't possible and it is important to specify a predicate unambiguously, the form *name/arity* is used. For example, the two predicates would be referred to as `pred/2` and `pred/1` respectively.

Now some useful facilities:

- `listing(Pred)` displays all the clauses whose head is `Pred`, e.g. `listing(pred)` displays all clauses headed by `pred`. If you have two (different) definitions of `pred` resident in memory, one with 1 argument and one with 2, then `listing(pred)` will display both. However, `listing(pred/2)` will only display the clauses headed by `pred` whose arity is 2.

- `trace` shows flow of control for your next query. Note it is only in operation for your very next query, then it is off. It will step you through the execution of that query. You type return to get it to move on one step at a time.

- `spy` does the same as `trace`. However this stays on until you switch it off. You switch it off using `nospy`. It is also more selective than `trace`. You have to give `spy` and `nospy` the name of a predicate, e.g. `spy(pred)`, `nospy(pred)` or `spy(pred/2)`, etc.

If you do use, `trace` and/or `spy`, then you will see how Prolog executes queries. This may not make full sense until we have discussed Prolog's procedural semantics. However, a few comments will probably make these facilities useful enough:

- Given a query, Prolog tries to match the query with a fact or the head of a rule in the database. It searches for a match from top to bottom of the database. So, if the goal to be proved is `p(X)`, and there are facts `p(a)`, `p(b)` and a rule `p(Y) :- q(Y)` in that order, then the first solution found will be `X = a`.

- When Prolog matches a goal with the head of a rule, then the subgoals in the body of the rule must also be proved for the whole rule to succeed. Prolog tries to solve these goals in the order in which they appear from left-to-right. For example, if Prolog is trying to prove a rule `p(X) :- q(X), r(X)`, then it will try to prove `q(X)` before proving `r(X)`.

□ Try out the tracing facilities NOW. They could be very useful in the next worksheets. For example, why not trace the execution of `above1, above2, above3, above4` and `above5`, and compare what Prolog is doing in each.

Acknowledgements

- The connected graph example is from Sterling & Shapiro.

- The other examples and exercises are either standard and commonplace or of my own invention.