# Lecture 5.2: Logic programming

In most general-purpose programming languages, control flow in a program is usually *explicit*, in that it's clear on a given point in the code what the next piece of code executed will be. For example, when I write the following Lua code:

```
local x = 3
local y = 4

for i = 1, x + y do
  print(i)
end
```

We understand that the code executes in sequential order, moving from the first line to the second line, then looping around the 'for' a few times. However, there's a long-lost programming paradigm called *logic programming* where this is not the case, where most control flow is *implicit* and defined by the implementation of the interpreter.

## 1. Basics

First, let's introduce the basics of the logic programming (LP) paradigm. LP stems from the realm of mathematical logic, so we can start by understanding some formal underpinnings. In LP, there is a set $O$ of all the objects (or "atoms") in the universe of the program. For example, if our objects our people, we could say $O = \{\texttt{john}, \texttt{mary}, \texttt{bethany}\}$. An $n$-ary relation on $O$ is a subset $R \subseteq O^n$ where $O^n$ is the $n$-th cross-product of the object set $O$. For example, a binary relationship like $\texttt{likes}(X, Y)$ has the property: $\texttt{likes}(X, Y) \subseteq O^2 = \{\{\texttt{john}, \texttt{john}\}, \{\texttt{john}, \texttt{mary}\}, \ldots\}$.

A logic program consists of a series of facts and rules for deducing further facts. For example:

```
likes(john, mary).
likes(mary, bethany).
likes(X, Y) :- likes(Y, X).
likes(X, Z) :- likes(X, Y), likes(Y, Z).
```

This program declares two facts, indicated syntactically by writing down a relation with atoms (always lowercase) as arguments. (Note that a period is always used to separate logical statements.) A fact declares that a certain set of objects is part of a particular relation.

Then we declare two rules, that allow us to deduce new facts. When multiple rules define the same relation, this is like an OR–it says "the facts in the likes relation are defined by the sum of what is derivable from these rules." These rules use *variables* like $X$ and $Y$ (always upper-case) to represent placeholders for objects. A rule is read as: if the relations on the right of the `:-`[1] hold, then the relation on the left holds. The comma is read like an "and".[2]

Here, the first rule says that `likes` is a symmetric relation, i.e. $\{Y, X\} \in \texttt{likes} \implies \{X, Y\} \in$

[1] Fun fact: the `:-` symbol used to be called a "dog's bollocks." Seriously.

[2] Here, we're discussing a specific subset of logic programming associated with the Datalog language. There are other logic programming languages like Prolog which contain a richer set of logical primitives.

likes. The second rule defines that likes is a transitive relation, i.e. $\{X, Y\} \in \texttt{likes} \wedge \{Y, Z\} \in \texttt{likes} \implies \{X, Z\} \in \texttt{likes}$. Note that the premises of rules are allowed to introduce new variables not bound in the "head" of the rule (the relation on the left, or the conclusion). The way the rules are written is canonical for the logic programming style, but we could also present them in a manner more familiar to you:

$$\frac{}{\texttt{likes}(\texttt{john},\texttt{mary})} \qquad \frac{}{\texttt{likes}(\texttt{mary},\texttt{bethany})} \qquad \frac{\texttt{likes}(Y, X)}{\texttt{likes}(X, Y)} \qquad \frac{\texttt{likes}(X, Y) \qquad \texttt{likes}(Y, Z)}{\texttt{likes}(X, Z)}$$

The logical semantics are the same in either presentation. Given these base facts and inference rules, we can use them to deduce new facts. For example, we can deduce:

```
likes(john, bethany).  % transitivity
likes(mary, john).     % symmetry
```

Now that we have a large database of facts, the last step is to define a way to query those facts. Queries in logic programming also take the form of relations, but they leave 0 or more entries in the relation as variables. For example, the query `likes(john, X)?` can be read as: "what are all the values of $X$ for which John likes $X$?" If you'd like to try to run some of these queries yourself, you can download a Datalog engine here: http://abcdatalog.seas.harvard.edu/

## 2. Unification

To answer queries like `likes(john, X)?`, one possible implementation of a query engine would check the query against every deducible fact in the fact database to see if they "match". Intuitively, `likes(john, X)` should match with the fact `likes(john, mary)` but not match with the fact `likes(mary, bethany)`. We can formalize this notion of "matching" with a concept called *unification*.

Unification is a procedure that attempts to produce a substitution that makes two terms equal. Substitution works just like it did in the lambda calculus, e.g.

$$[X \to \texttt{mary}]\ \texttt{likes}(\texttt{john}, X) = \texttt{likes}(\texttt{john},\texttt{mary})$$

Generally, unification attempts to produce a *most general unifier*, which is the minimal number of substitutions requires to make two terms equal (note that we always prefer to map variables to atoms before mapping them to other variables). For example:

$$\texttt{unify}\ []\ \texttt{likes}(X, Y)\ \texttt{likes}(\texttt{john},\texttt{mary}) = [X \to \texttt{john}, Y \to \texttt{mary}]$$

Here, unify is a function that takes three arguments: an existing substitution environment (empty in the above example), and then two relations containing a mix of variables and atoms. Terms do not have to unify–for example:

$$\texttt{unify}\ []\ \texttt{likes}(\texttt{bethany}, Y)\ \texttt{likes}(\texttt{john},\texttt{mary})$$

Here, these terms will never be equal because $\texttt{bethany} \neq \texttt{john}$, and we never substitute an atom for another atom. This would be called a unification failure. The procedure for running a query against a fact database, then, is to attempt to unify it against each of our facts, returning the fact as a matching result if unification succeeds.

## 3. Applications

Let's explore a few more applications to get a feel for the expressive power of the logic programming before discussing the implementation. A canonical example is exploring family trees:

```
parent(john, mary).
parent(bethany, john).
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

Given a parent relation, grandparenting is just transitive on parenting.

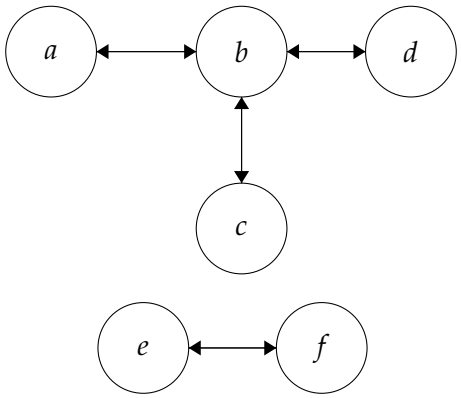
```
married(bethany, luke).
parent(X, Z) :- married(X, Y), parent(Y, Z)
```

We can extend relations like parent to use auxiliary information like the married relation here, which says: if $Y$ is the parent of $Z$ and $X$ is married to $Y$, then $X$ should be the parent of $Z$ as well even if we didn't explicitly write that down as a fact. Here, we could derive that `parent(luke, john)`. As an exercise to the reader, how would you define the rule(s) for cousins? Nephews? Relatives?

Another cool application also has to do with graphs. Consider a graph defined by its edges:

```
edge(a, b).
edge(b, c).
edge(b, d).
edge(e, f).
edge(X, Y) :- edge(Y, X).  % undirected edges
```

This corresponds to the following graph:




A common property to understand when dealing with graphs is connectedness—we want to know for each node, which components of the graph it's in. Now imagine for a moment that I asked you to implement an algorithm for graph connectedness in your favorite imperative or functional language. How would you do it? You would probably have to write some kind of graph search algorithm—either breadth-first search or depth-first search, doesn't matter which here. If you've implemented these algorithms before, you would know that this is not trivial! Your algorithm has to maintain a stack/queue of all the nodes, track which ones you've seen/not seen, find frontiers of nodes, and so on.

This is a natural property of using languages with explicit control flow, which is that when faced with a problem, you have to tell the language *how* to solve the problem. By contrast, in the logic programming paradigm, you don't tell the interpreter *how* to solve the problem, but instead *what* the problem is. Specifically, here's our connectedness algorithm in :

```
connected(X, Y) :- edge(X, Y).
connected(X, Z) :- connected(X, Y), connected(Y, Z).
```

Wow! Two lines of code! So easy. Here, all we had to do was specify the logical properties of connectedness—i.e. that two nodes are connected either if there is a single edge between them, or if there is some node that is connected to both of them. The runtime, then, is free to decide how to implement the search procedures that underlies connectedness–it could be breadth-first, depth-first, or something simpler/smarter. Because we separated the *specification* of the problem from its *implementation*, it allows us to both simplify our code and open a space of optimizations in determining the most efficient way to solve our problem.[3]

## 4. Implementation

How would one go about implementing a Datalog interpreter? There are many possible implementatinos, but we will discuss the simplest (and subsequently least efficient) one. Constructing a fact database has two main steps: setup and query, where setup takes a program (facts and rules) and builds a data structure for use in the query phase, where the user issues multiple relational queries against the precomputed fact database.

In our simple implementation, the setup phase will exhaustively deduce every possible fact from our inputs. A query, then, is simply a matter of matching against all known facts. The process of deriving all known facts is called *saturation*. Saturation means deriving a set of new facts from a set of current facts, and repeating that process until no further facts can be derived. For example, consider the likes example from before:

```
likes(john, mary).
likes(mary, bethany).
likes(X, Y) :- likes(Y, X).
likes(X, Z) :- likes(X, Y), likes(Y, Z).
```

In one step of saturation, we start with a set of facts, in this case what the user has provided us: $S = \{\texttt{likes(john,mary)}, \texttt{likes(mary,bethany)}\}$. Then for each rule, we determine what new facts can be deduced from our current facts. For example, let's take the rule:

```
likes(X, Z) :- likes(X, Y), likes(Y, Z).
```

Our naive method of deduction is to consider every possible combination of facts that *might* satisfy the premises. In the case of the above rule, for each relation in the premise, we consider all known facts for that relation as candidates to satisfy it. A candidate for the whole rule, then is the cross

[3]However, logic programming languages usually just have one way to do the search and that method can be inefficient, so optimizing these problems for large graphs or many rules can get tricky. Disallowing the user to specify these implementation details often entails such a tradeoff.

product of the candidates for each relation, which here would be:

$$S \times S = \{\{\texttt{likes}(\texttt{john}, \texttt{mary}), \texttt{likes}(\texttt{john}, \texttt{mary})\}, \{\texttt{likes}(\texttt{john}, \texttt{mary}), \texttt{likes}(\texttt{mary}, \texttt{bethany})\}, \ldots\}$$

For each candidate for the rule, we then consider if that candidate satisfies the premises of the rule. To do this, we return to the idea of unification: for each relation in the premises of the rule, we attempt to unify the corresponding fact with the relation. If this succeeds, we accumulate the substitution environment across all the rules, and if it fails, we toss the candidate.

For example, consider the first candidate in the above list, $\{\texttt{likes}(\texttt{john}, \texttt{mary}), \texttt{likes}(\texttt{john}, \texttt{mary})\}$. We can unify the first fact with the first relation in the transitivity rule:

$$\texttt{unify}\ [] \ \texttt{likes}(\texttt{john}, \texttt{mary})\ \texttt{likes}(X, Y) = [X \to \texttt{john}, Y \to \texttt{mary}]$$

That works, and produces a valid substitution environment. However, we then need to check the next relation with the environment we just created:

$$\texttt{unify}\ [X \to \texttt{john}, Y \to \texttt{mary}]\ \texttt{likes}(\texttt{john}, \texttt{mary})\ \texttt{likes}(Y, Z)$$

This will produce a unification failure. In order for these terms to unify, we would have to map $Y \to \texttt{john}$, however this conflicts with our existing substitution environment that says $Y \to \texttt{mary}$. Therefore we deduce that this candidate does not apply to the transitivity rule, and throw it out. Let's try the next candidate, $\{\texttt{likes}(\texttt{john}, \texttt{mary}), \texttt{likes}(\texttt{mary}, \texttt{bethany})\}$:

$$\texttt{unify}\ [] \ \texttt{likes}(\texttt{john}, \texttt{mary})\ \texttt{likes}(X, Y) = [X \to \texttt{john}, Y \to \texttt{mary}]$$

As before, this produces the same substitution environment. However, when we go to check the next relation in the rule:

$$\begin{aligned}\texttt{unify}\ [X \to \texttt{john}, Y \to \texttt{mary}]\ \texttt{likes}(\texttt{mary}, \texttt{bethany})\ \texttt{likes}(Y, Z)& \\ = [X \to \texttt{john}, Y \to \texttt{mary}, Z \to \texttt{bethany}]&\end{aligned}$$

The unification succeeds! This means that the candidate $\{\texttt{likes}(\texttt{john}, \texttt{mary}), \texttt{likes}(\texttt{mary}, \texttt{bethany})\}$ is a valid premise for the transitivity rule, which should match our intuition (John likes Mary, and Mary likes Bethany, so by transitivity John likes Bethany). The last step is to geenerate our new fact, which we get by applying the final substitution environment to the relation at the head of the rule:

$$[X \to \texttt{john}, Y \to \texttt{mary}, Z \to \texttt{bethany}]\ \texttt{likes}(X, Z) = \texttt{likes}(\texttt{john}, \texttt{bethany})$$

We can add this new fact $\texttt{likes}(\texttt{john}, \texttt{bethany})$ to our fact database and continue repeating this process for every candidate and for every rule until no new facts can be generated. A full pseudocode description of the algorithm is available in the assignment 5 handout. After that process, we now have a complete database of all deducible facts. As mentioned above, a query then is attempting to unify the input against all facts in the database, returning those that succesfully unify.