A Brief Introduction to Basic Logic Programming

Michael Genesereth
CodeX: The Stanford Center for Legal Informatics
Stanford University
Logic Programming is a style of programming in which programs take the form of sentences in the language of Symbolic Logic.

\[ p(a,b) \quad q(b,c) \]

\[ \neg p(b,d) \quad \forall x. \forall y. (p(x,y) \implies q(x,y)) \]

\[ p(c,b) \lor p(c,d) \quad \exists x. p(x,d) \]

Logic Programming is a style of programming based on Symbolic Logic. In Logic Programming, the programmer writes programs in the form of sentences in the language of Symbolic Logic. Collections of such sentences are called logic programs. In some cases, the sentences in a logic program simply define concepts relevant to the application area of the program; in other cases, they also capture the dynamics of the application area and they encode constraints on the behavior of agents operating in that application area.
Logic programs can be thought of in two ways - the declarative and the procedural interpretations. First of all, as formal representations of information about an application area, they can be read and analyzed by humans. In this respect, they are similar to laws written in English but without the ambiguities of natural language.
A second benefit of using logic to express laws is that the resulting logic programs can be read and analyzed by computers. In the legal domain, this leads to computer systems capable of doing legal calculations, such as compliance checking, legal planning, regulatory analysis, and so forth.
Over the years, numerous variations of Logic Programming have been explored (e.g. Abductive Logic Programming, Inductive Logic Programming, and Answer Set Programming), and a variety of different Logic Programming languages have been developed (e.g. Datalog and Prolog and Golog). In this session, we focus on an elementary form of Logic Programming, called Basic Logic Programming, and we describe a corresponding language, called Epilog.
Today, we are going to concentrate on the classic use of basic logic programs, viz. the application of logical rules to facts to produce additional facts. In the legal context, the logic program encodes applicable rules and regulations, the facts describe the details of a given case, and the conclusions are the legal analysis of that case. That said, to make things easy, we will be looking at non-legal examples. Once you learn the language, your mission will be to apply it in legal settings to perform legal analysis.
Databases
If you are like me, when you think about the world, you think in terms of objects and relationships among these objects. Objects include things like people and offices and buildings. Relationships include things like the parenthood, ancestry, office assignments, office locations, and so forth.

For most of us, conceptualizing the world is a subconscious process. We do not even think about it. And we usually assume our initial way of looking at the world is the only one that makes sense. But this is not always the case. Sometimes, it is useful to conceptualize the world in different ways.
Let's look at an example drawn from the world of genealogy. We start with data about parentage. Art is the parent of Bob, Bob is the parent of Cal and Cam, and so forth.
Of course, these are not the only relationships we can consider. We can talk about the ancestor relation. One person is an ancestor of another iff the first person is a parent of the second or a parent of a parent, and so forth. Or we can talk about the sibling relation.
Object constants and relation constants consist of lower case letters, digits, underscores. No difference in spelling between the two; type of constant is determined by its use.

<table>
<thead>
<tr>
<th>Object Constants:</th>
<th>Relation Constants:</th>
</tr>
</thead>
<tbody>
<tr>
<td>art</td>
<td>parent</td>
</tr>
<tr>
<td>bob</td>
<td>grandparent</td>
</tr>
<tr>
<td>bea</td>
<td>sibling</td>
</tr>
<tr>
<td>cal</td>
<td>ancestor</td>
</tr>
<tr>
<td>cam</td>
<td>related</td>
</tr>
<tr>
<td>coe</td>
<td></td>
</tr>
<tr>
<td>cory</td>
<td></td>
</tr>
</tbody>
</table>

The vocabulary of a database is a collection of object constants and relation constants.
A datum / fact is an expression formed from a relation constant and zero or more object constants enclosed in parentheses and separated by commas.

\[ \text{parent}(\text{art}, \text{bob}) \]
A dataset is any set of data that can be formed from the vocabulary of a database. Intuitively, we can think of the data in a dataset as the facts that we believe to be true in the world; data that are not in the dataset are assumed to be false.

As an example of these concepts, consider a small interpersonal database. The objects in this case are people. The relationships specify properties of these people and their interrelationships.
A dataset is any set of data that can be formed from the vocabulary of a database. Intuitively, we can think of the data in a dataset as the facts that we believe to be true in the world; data that are not in the dataset are assumed to be false.

As an example of these concepts, consider a small interpersonal database. The objects in this case are people. The relationships specify properties of these people and their interrelationships.
Datasets

A dataset is any set of data that can be formed from the vocabulary of a database. Intuitively, we can think of the data in a dataset as the facts that we believe to be true in the world; data that are not in the dataset are assumed to be false.

As an example of these concepts, consider a small interpersonal database. The objects in this case are people. The relationships specify properties of these people and their interrelationships.
A dataset is any set of data that can be formed from the vocabulary of a database. Intuitively, we can think of the data in a dataset as the facts that we believe to be true in the world; data that are not in the dataset are assumed to be false.

As an example of these concepts, consider a small interpersonal database. The objects in this case are people. The relationships specify properties of these people and their interrelationships.
A dataset is any set of data that can be formed from the vocabulary of a database. Intuitively, we can think of the data in a dataset as the facts that we believe to be true in the world; data that are not in the dataset are assumed to be false.

As an example of these concepts, consider a small interpersonal database. The objects in this case are people. The relationships specify properties of these people and their interrelationships.
The **arity** of a relation constant is the number of arguments that can be associated with the relation constant in writing complex expressions in the language.

Facts involving **unary** Relations (1 argument):
- `male(bob)`

Facts involving **binary** Relations (2 arguments):
- `parent(art,bob)`

Facts involving **ternary** Relations (3 arguments):
- `prefers(art,bob,bea)`

Each relation constant has an associated arity, i.e. the number of objects involved in any instance of the corresponding function or relation. Unary relations are those that take 1 argument; binary relation constants take two arguments; ternary relation constants take three arguments. Beyond that, we often say that the constants are n-ary. Note that it is possible to have a relation constant with no arguments - a relation that is simply true or false.
The order of arguments in an instance of a relation is determined by one’s understanding of the relation.

Example:

\[
\text{prefers(art, bob, bea)}
\]

For me, this sentence means that Art prefers Bea to Bob. Other interpretations are possible; the important thing is to be consistent - once you choose, stick with it.

Note on Order of Arguments

The order of arguments in such sentences is arbitrary. Given the meaning of the prefers relation in our example, the first argument denotes the subject, the second argument is the person who is preferred, and the third argument denotes the person who is less preferred. We could equally well have interpreted the arguments in other orders. The important thing is consistency – once we choose to interpret the arguments in one way, we must stick to that interpretation everywhere.
The spelling of constants carries no meaning (except as informal documentation).

Example:

\[ \text{prefers(art,bob,bea)} \]
\[ \text{versus} \]
\[ \text{coulish(widget,gadget,framis)} \]

The meaning of a constant is determined by the rules that mention it.

Note also that the spelling of constants has no meaning, except perhaps as documentation. The *meaning* of a constant is determined by the rules that define it.
Rules
Consider this kinship dataset. As situation as before. One thing to notice is that it is highly redundant. The information about parentage actually determines the information about sibling and grandparenthood. Rules provide us with a way to express these relationships in general; and, once, we have these rules, we can delete the redundant data, safe in the knowledge that we can recompute it as necessary.
Solution

Represent information as combination of data and rules. Encode information about “base” relations as **facts** in a dataset and write **rules** to define “view” relations in terms of the base relations.

Benefits:
- Economy - fewer facts need to be stored
- Less chance of getting out of sync
- Rules work for any number of objects (think numbers)

Consider this kinship dataset. As situation as before. One thing to notice is that it is highly redundant. The information about parentage actually determines the information about sibling and grandparenthood. Rules provide us with a way to express these relationships in general; and, once, we have these rules, we can delete the redundant data, safe in the knowledge that we can recompute it as necessary.
In our kinship example, we can reduce our dataset to just parent facts and can then write rules to define all of the other relations.
Object Constants
  art  bob  bea

Relation Constants
  parent  sibling  grandparent

Variables
  X  Y23  Plaintiff

As before, object constants and relation constants consist of lower case letters, digits, underscores. Variables begin with a capital letter and, otherwise, are spelled the same as constants.

The vocabulary of a logic program is the same as that of databases, except that we now have variables as well as constants. As we shall see in a moment, a variable is a way to state properties of any arbitrary object without explicitly naming the objects. As before, we write symbols as strings of letters, digits, and a few non-alphanumeric characters (e.g. "_"), Constants must begin with a lower case letter or digit. Variables must begin with an upper case letter. A term is either an object constant or a variable.
A term is either an object constant or a variable.

art  bob  X  Y23

NB: Epilog also allows users to write more complex terms. And this significantly increases its representational power. Not needed for our purposes today.

The vocabulary of a logic program is the same as that of databases, except that we now have variables as well as constants. In what follows, we write such symbols as strings of letters, digits, and a few non-alphanumeric characters (e.g. "_"). Constants must begin with a lower case letter or digit. Variables must begin with an upper case letter. A term is either an object constant or a variable.
Atoms

\( p(a,b), \ p(a,X), \ p(Y,c) \)

Negations

\( \neg p(a,b) \)

A literal is either an atom or a negation of an atom.

\( p(a,Y), \ \neg p(a,Y) \)

An atom is a positive literal.
A negation is a negative literal.

An atom is an expression consisting of an n-ary relation constant and n terms. In what follows, we write atoms in traditional mathematical notation - the relation constant followed by its arguments enclosed in parentheses and separated by commas. For example, if \( p \) is a binary relation constant and if \( a \) and \( b \) are object constants, then \( p(a,b) \) is an atom. A negation is an expression formed using the negation sign \( \neg \) and an atom. For example, \( \neg p(a,b) \). A literal is either an atom or a negation. An atom is sometimes called a positive literal, and a negation is sometimes called a negative literal.
## Rules

**Rule Syntax:**

```
\( r(X,Y) :– p(X,Y) \land \neg q(Y) \)
```

Where `\( r(X,Y) \)` is the head, `\( p(X,Y) \)` and `\( \neg q(Y) \)` are subgoals.

**Rule Semantics (intuitive):**

For any instance of a rule, the head is true if

1. all positive literals in the body are true and
2. all negative literals in the body are false.

---

A rule is an expression consisting of a distinguished atom, called the head, and a conjunction of zero or more literals, called the body, separated by the `:–` operator. The literals in the body are called subgoals.

The intended meaning is that an instance of the head is true whenever corresponding instances of all of the positive subgoals are true and all of the negative subgoals are false.
The rule here states that Art is the grandparent of a person Cal if Art is the parent of a Bob and Bob is the parent of Cal. Since both of the subgoals in this case are true, we know that the head is true as well.
Here is another example. The rule states that Art is the grandparent of a person Coe if Art is the parent of a Bob and Bob is the parent of Coe. Since both of the subgoals once again are true, we know that the head is true as well.
<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| parent(art,bob)  
parent(art,bea)  
parent(bob,cal)  
parent(bob,coe)  

+  
grandparent(art,cal):=parent(art,bob)&parent(bob,cal)  
grandparent(art,coe):=parent(art,bob)&parent(bob,coe)  

=  
grandparent(art,cal)  
grandparent(art,coe)  

Putting both of these rules together, we can drive both conclusions from the same givens.
Note that, in general, we may need to consider other possibilities as well when writing our rules. While this works, it is very verbose. Fortunately, there is a better way. Using variables, we can collapse all of these rules into one single rule that applies in all cases.
The rule here defines the grandparent relation in terms of the parent relation. A person X is the grandparent of a person Z if X is the parent of a person Y and Y is the parent of Z. The variable Y here is a "thread variable" that connects the first subgoal to the second but does not itself appear in the head of the rule.
A logic program is simply a collection of rules of the sort we have just seen. For example, we can define a variety of different kinship facts as shown here.
Exercises
As an exercise in logic programming, once again consider our kinship applications. Suppose we were to start with information about the parent relation and the gender relations male and female. The objective here is to write rules that define various kinship relations in terms of these base relations.
Let's start by defining a person to be a male or a female.
**Example**

## Personhood:

<table>
<thead>
<tr>
<th>Data:</th>
<th>View:</th>
</tr>
</thead>
<tbody>
<tr>
<td>male(art)</td>
<td>person(art)</td>
</tr>
<tr>
<td>male(bob)</td>
<td>person(bob)</td>
</tr>
<tr>
<td>male(cal)</td>
<td>person(cal)</td>
</tr>
<tr>
<td>male(cam)</td>
<td>person(cam)</td>
</tr>
<tr>
<td>female(bea)</td>
<td>person(bea)</td>
</tr>
<tr>
<td>female(coe)</td>
<td>person(coe)</td>
</tr>
<tr>
<td>female(cory)</td>
<td>person(cory)</td>
</tr>
</tbody>
</table>

Note that the same relation can appear in the head of more than one rule. For example, the person relation is true of a person Y if there is an X such that X is the parent of Y *or* if Y is the parent of some person Z. Note that in this case the conditions are disjunctive (at least one must be true), whereas the conditions in the grandfather case are conjunctive (both must be true).
Ancestors:

<table>
<thead>
<tr>
<th>Data:</th>
<th>View:</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent(art,bob)</td>
<td>ancestor(art,bob)</td>
</tr>
<tr>
<td>parent(art,bea)</td>
<td>ancestor(art,bea)</td>
</tr>
<tr>
<td>parent(bob,cal)</td>
<td>ancestor(bob,cal)</td>
</tr>
<tr>
<td>parent(bob,coe)</td>
<td>ancestor(bob,coe)</td>
</tr>
<tr>
<td>parent(bea,coe)</td>
<td>ancestor(bea,coe)</td>
</tr>
<tr>
<td>parent(bea,cory)</td>
<td>ancestor(bea,cory)</td>
</tr>
<tr>
<td></td>
<td>ancestor(art,cal)</td>
</tr>
<tr>
<td></td>
<td>ancestor(art,cam)</td>
</tr>
<tr>
<td></td>
<td>ancestor(art,coe)</td>
</tr>
<tr>
<td></td>
<td>ancestor(art,cory)</td>
</tr>
</tbody>
</table>

Now let's define ancestor in terms of parenthood.
A person X is an ancestor of a person Z if X is the parent of Z or if there is a person Y such that X is an ancestor of and Y is an ancestor of Z. This example shows that it is possible for a relation to appear in its own definition. (But recall our discussion of stratification for a restriction on this capability.)

### Example

**Ancestors:**

\[
\text{ancestor}(X, Z) := \text{parent}(X, Z) \\
\text{ancestor}(X, Z) := \text{parent}(X, Y) \land \text{ancestor}(Y, Z)
\]

**Data:**

- `parent(art,bob)`
- `parent(art,bea)`
- `parent(bob,cal)`
- `parent(bob,coe)`
- `parent(bea,coe)`
- `parent(bea,cory)`

**View:**

- `ancestor(art,bob)`
- `ancestor(art,bea)`
- `ancestor(bob,cal)`
- `ancestor(bob,coe)`
- `ancestor(bea,coe)`
- `ancestor(bea,cory)`
- `ancestor(art,cal)`
- `ancestor(art,cam)`
- `ancestor(art,coe)`
- `ancestor(art,cory)`

---
Finally, let's look at the problem of defining the aspirant relation, which holds of a person if and only if that person does not have any children.

<table>
<thead>
<tr>
<th>Data:</th>
<th>View:</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent(art,bob)</td>
<td>childless(cal)</td>
</tr>
<tr>
<td>parent(art,bea)</td>
<td>childless(cam)</td>
</tr>
<tr>
<td>parent(bob,cal)</td>
<td>childless(coe)</td>
</tr>
<tr>
<td>parent(bob, cam)</td>
<td>childless(cory)</td>
</tr>
<tr>
<td>parent(bea, coe)</td>
<td></td>
</tr>
<tr>
<td>parent(bea, cory)</td>
<td></td>
</tr>
</tbody>
</table>
A childless person is one who has no children. We can define the property of being childless with the rules shown below. The first rule states that a person X is childless if X is a person and it is not the case that X is a parent. The second rule says that isparent is true of X if X is the parent of some person Y.

Note the use of the helper relation isparent here. It is tempting to write the childless rule as childless(X) :- person(X) & ~parent(X,Y). However, this would be wrong. This would define X to be childless if X is a person and there is "some" Y such that X is ~parent(X,Y) is true. But we really want to say that ~parent(X,Y) holds for all "Y. Defining isparent and using its negation in the definition of childless allows us to express this "universal quantification".
Safety
Suppose we had a database in which \( p(a,b) \) is true. Then, the body of the first rule is satisfied if we let \( X \) be \( a \) and \( Y \) be \( b \). In this case, we can conclude that every corresponding instance of the head is true. But what should we substitute for \( Z \)? Intuitively, we could put anything there; but there could be infinitely many possibilities. For example, we could write any number there. While this is conceptually okay, it is practically problematic.
### Rule:

\[
t(X, Y) :- p(X, Y) \& \neg q(Y, Z)
\]

### Fact:

| p(a, b) | t(a, b) |
| p(a, c) | t(a, c) |
| q(c, d) |        |

### Output:

- \( p(a, b) \)
- \( p(a, c) \)
- \( q(c, d) \)

### Alternative Rules:

\[
\begin{align*}
t(X, Y) & :- p(X, Y) \& \neg qqq(Y) \\
qqq(Y) & :- q(Y, Z)
\end{align*}
\]

Suppose we had a database with just two facts, viz. \( p(a, b) \) and \( q(b, c) \). In this case, if we let \( X \) be \( a \) and \( Y \) be \( b \) and \( Z \) be anything other than \( c \), then both subgoals true, and we can conclude \( t(a, b) \). The main problem with this is that many people incorrectly interpret that negation as meaning there is no \( Z \) for which \( q(Y, Z) \) is true, whereas the correct reading is that \( q(Y, Z) \) needs to be false for just one binding of \( Z \). As we have seen before, there is a simple way of expressing this other meaning without writing unsafe rules.
A rule is *safe* if and only if every variable in the head appears in some positive subgoal in the body.

**Safe Rule:**
\[ r(X, Z) :- p(X, Y) \land q(Y, Z) \land \neg r(X, Y) \]

**Unsafe Rule:**
\[ r(X, Z) :- p(X, Y) \land q(Y, X) \]

**Unsafe Rule:**
\[ r(X, Y) :- p(X, Y) \land \neg q(Y, Z) \]

*All rules must be safe.*

The first of these restrictions is called safety. A rule in a logic program is safe if and only if every variable that appears in the head or in any negative literal in the body also appears in at least one positive literal in the body. The first rule shown here is safe. Variables X and Z appear in the head and Y appears in a negative subgoal. Fortunately, all three of those variables appear in positive subgoals as well, and so the rule is safe. The second is not safe because variable Z appears in the head but not in any positive subgoal. The third rule is not safe because the variable Z appears in a negative subgoal but not in a positive subgoal. In deductive databases, we require all rules to be safe.
Stratified Negation
The problem with unstratified logic programs is that there is a potential ambiguity. As an example, consider the program above and assume we had a database containing p(a,b), p(b,a), q(a), and q(b). From these facts we can conclude r(a,b) and r(b,a) are both true. So far so good. But what can we say about s? If we take s(a,b) to be true and s(b,a) to be false, then the second rule is satisfied. If we take s(a,b) to be false and s(b,a) to be true, then the second rule is again satisfied. We can also take them both to be true. The upshot is that there is ambiguity about s. By concentrating exclusively on programs that are stratified with respect to negation, we avoid such ambiguities.
The dependency graph for a set of rules is a directed graph in which (1) the nodes are the relations mentioned in the head and bodies of the rules and (2) there is an arc from a node $p$ to a node $q$ whenever $p$ occurs with the body of a rule in which $q$ is in the head.

\[
\begin{align*}
  r(X,Y) & : = p(X,Y) \ & \& q(X,Y) \\
  s(X,Y) & : = r(X,Y) \\
  s(X,Z) & : = r(X,Y) \ & \& t(Y,Z) \\
  t(X,Z) & : = s(X,Y) \ & \& s(Y,X)
\end{align*}
\]

A set of rules is recursive if it contains a cycle. Otherwise, it is non-recursive.

The next two restrictions on GDL descriptions have to do with recursion. The restrictions are best defined using the notion of dependency graphs. The dependency graph for a set of rules is a directed graph in which (1) the nodes are the relations mentioned in the head and bodies of the rules and (2) there is an arc from a node $p$ to a node $q$ whenever $p$ occurs with the body of a rule in which $q$ is in the head. A set of rules is recursive if and only if its dependency graph contains a cycle.
The negation in a set of rules is said to be *stratified* if and only if there is no recursive cycle in the dependency graph involving a negation.

**Stratified Negation:**

\[
\begin{align*}
  r(X, Z) &:= p(X, Y) \\
  r(X, Z) &:= r(X, Y) & \land & r(Y, Z)
\end{align*}
\]

**Negation that is not stratified:**

\[
\begin{align*}
  r(X, Z) &:= p(X, Y) \\
  r(X, Z) &:= p(X, Y) & \land & \neg r(Y, Z)
\end{align*}
\]

*All negations must be stratified.*

A negation in a set of rules is said to be stratified if and only if there is no recursive cycle in the dependency graph involving a negation. For example, the first rule set shown here is not stratified because there is a cycle involving a negative occurrence of \( r \). By contrast, the second set of rules is stratified. The rule set is recursive, but there is no negation in the cycle. The only negative occurrence of \( r \) occurs in the definition of \( t \) and is not part of any recursion. In deductive databases, we require all negations to be stratified.
Extensions
Pre-defined concepts:
- Arithmetic Functions (+, *, min, max)
- String functions (concatenate)
- Comparison Operators (> , < , =)

NB: operators are spelled out (e.g. plus for +, times for *)
NB: n-ary functions represented as (n+1)-ary relations

Example:

\[
\text{combage}(X,Y,S) :- \text{age}(X,M) \land \text{age}(Y,N) \land \text{plus}(M,N,S)
\]

NB: unbound variables allowed in last arg of function only
NB: all variables of comparisons must be bound

In many practical logic programming languages, mathematical functions are represented as relations. For example, the binary addition operator + is often represented by the ternary relation constant plus. For example, the rule shown here defines the combined age of two people.
Aggregate Operators

Aggregates:
  countofall
  sumofall
  avgofall

Example:

    numgrandchildren(X,N) :-
      person(X) &
      countofall(Z,grandparent(X,Z),N)

NB: Only variables allowed in condition are aggregation variable or variables bound in other positive subgoals.

There are also aggregate operators such as countofall, sumofall, and avgofall. For example the rule here defines the number of a person's grandchildren using the countofall operator. N is the number of grandchildren of X if N is the count of all Z such that X is the grandparent of Z.