There are several distinct but often equivalent ways to think about logic programs:

1. As computing logical consequences of the program.

2. As describing a search procedure to be followed (by SLD resolution): e.g.

   \[
   A : - B, C, D. \\
   A : - E, F. \\
   A : - G. 
   \]

   says to solve \( A \), first do \( B \), then \( C \), then \( D \). If that fails, try \( E \) then \( F \). If that fails, try \( G \).

3. As computing what is true in a particular first order structure, derived by applying the algorithm HORN to the set of ground instances of the program.
The “minimal model” of a logic program

Let \( P \) be a logic program and let \( H \) be the set of all ground atoms that are marked when the algorithm HORN is run on the set of all ground instances of \( P \). If the computation does not terminate, take \( H \) to be the set of all ground atoms that are marked at any stage of the computation. Define the first order structure \( \mathcal{M} \) as follows:

1. The universe \( A^\mathcal{M} \) of \( \mathcal{M} \) is the set of all ground terms that can be constructed using the constants and function symbols in \( P \).
2. For each predicate \( p \) in \( P \), the interpretation \( p^\mathcal{M} \) of \( P \) is the set of all tuples \((t_1, \ldots, t_n)\) such that \( p(t_1, \ldots, t_n) \) is in \( H \).
3. For each constant symbol \( c \) in \( P \), the interpretation \( c^\mathcal{M} \) of \( P \) is \( c \).
4. For each function symbol \( f \) of arity \( n \) in \( P \), the interpretation \( f^\mathcal{M} \) is the function from \( n \) ground terms to ground terms defined by taking \( f^\mathcal{M}(t_1, \ldots, t_n) \) to be the ground term \( f(t_1, \ldots, t_n) \)

Some consequences of this definition:

1. For all ground terms \( t \), the interpretation of \( t^\mathcal{M} \) of \( t \) in \( \mathcal{M} \) is \( t \) itself.
2. If \( t_1 \) and \( t_2 \) are two distinct terms, then \( \mathcal{M} \models \neg(t_1 = t_2) \).
   (Compare this with the fact that \( \mathcal{M} \models 2 + 2 = 4 \) in a model for arithmetic.)
3. For all ground atoms \( p(t_1, \ldots, t_n) \), we have
   \( \mathcal{M} \models p(t_1, \ldots, t_n) \) if and only if \( p(t_1, \ldots, t_n) \in H \).
4. For all rules \( H : - B_1, \ldots, B_n \) of the program \( P \), if the variables in this rule are \( X_1, \ldots, X_n \) then
   \[ \mathcal{M} \models \forall X_1 \ldots \forall X_n ((B_1 \land \ldots \land B_n) \rightarrow H) \]
A predicate $p$ depends on a predicate $q$ in a logic program if there exists a rule of the form $H : \neg B_1, \ldots, B_n$ where $p$ is the predicate in the head $H$ and $q$ is the predicate in one of the atoms $B_i$ in the body.

Example: In

\[
\begin{align*}
p(X) :& \quad \neg q(Y, X), r(Y, Z). \\
p(X) :& \quad \neg s(X, X). \\
q(X, Y) :& \quad \neg s(X, X), r(X, Y). \\
r(a, b). \\
r(X, Y) :& \quad \neg s(X, Y). \\
s(b, b).
\end{align*}
\]

1. $p$ depends on $q$, $r$ and $s$
2. $q$ depends on $s$ and $r$
3. $r$ depends on $s$
4. $s$ depends on nothing.
A logic program is *recursive* if there exists a sequence of predicates \( p_0, p_1, \ldots, p_n \) such that

1. for each \( i \), \( p_{i+1} \) depends on \( p_i \), and
2. \( p_0 = p_n \).

(i.e. there exists a cycle in the dependency relation.)

The program above is not recursive. The *path* program is recursive, since *path* depends on itself.

Suppose that \( P \) is a non-recursive logic program containing no function symbols and \( B_1, \ldots, B_n \) are atoms and \( \theta \) is a ground substitution for the variables of \( B_1, \ldots, B_n \). Then

1. \( P \models (B_1 \land \ldots \land B_n)\theta \)
2. There exists an SLD (leftmost selection) refutation of \( P \cup \{\bot : - B_1 \land \ldots \land B_n\} \) computing an answer substitution \( \theta' \) that is at least as general as \( \theta \).
3. \( M \models (B_1 \land \ldots \land B_n)\theta \) where \( M \) is the minimal model of \( P \).
For many additional logic programs these three different views are also equivalent. However, in general, equivalence fails because of problem of infinite branches, lack of occurs check, and other impure features of logic programs.

Some more recently developed systems/languages, e.g.,

1. CORAL (U. Wisconsin)
2. Aditi (U. Melbourne)
3. XSB (New York, Stony Brook)

attempt to take a purer logical view than Prolog, while retaining its logical efficiency, but these remain research systems that have not yet been widely adopted.

Nevertheless, the three views are useful in helping to design Prolog programs.

When you design a program from the logical consequence or minimal model point of view, you are describing *what* to compute, rather than *how* to compute it. This means that it is often the case that your program can be used in many different *modes*. For example: provided the graph described in the edge facts has no cycles, the SLD computation correctly and completely answers the following queries:

1. path(X,Y)?
2. path(a,X)?
3. path(a,b)?
4. path(X,b)?
An *procedural* programming approach would typically develop the program with only one of these modes in mind, producing a program that is not likely to work for the others.

As a general rule in logic programming:

1. **first** describe what you want to compute logically.
2. **then** worry about whether the prolog procedural semantics is going to work correctly for the query modes you are interested in.

**Equality**

Computationally, the equals symbol corresponds to unification of terms in a logic program. Operationally, “=“ is a predicate just like any other, defined by the single clause

\[ X = X. \]

Examples:

: \( X = 3, X = 2? \)  
: \( X = f(Y), Y = a? \)

** no

\( X = f(a) \)

\( Y = a \)

Note that it is *not* possible to “assign a value to a variable” in a logic program. However, one can “make the value more and more precise” by a number of unification steps.
One consequence of the fact that in logic programming, two ground terms are equal only if they are the same term, is that we can use terms to capture data structures.

Some examples:

1. lists
2. trees
3. formulas

LISTS

A list \(a_1, a_2, \ldots, a_n\) can be represented as a term

\[ l(a_1, l(a_2, \ldots l(a_n, \text{emptylist}))) \]

using some binary function symbol \(l\) and a constant \(\text{emptylist}\).

Prolog has such a function symbol and constant hidden “under the hood” but uses a special notation for lists:

1. \([a_1, \ldots, a_n]\)
2. \([X \mid Y]\) — the list with first element \(X\) (the head) and \(Y\) (the tail of all the remaining elements).
3. \([]\) — the empty list.
: [2, 3] = [X|Y]?
X = 2
Y = [3]

: [2, 3] = [X, Y]?
X = 2
Y = 3

: [2, 3, 4|X] = [2, 3, 4, 5, 6]?
X = [5, 6]

: [2, 3] = [X, Y|Z]?
X = 2
Y = 3
Z = []

\[\text{concat}(X, Y, Z) \text{ is true if the result of concatenating list } X \text{ to list } Y \text{ is list } Z.\]

\[
\begin{align*}
\text{concat}([], X, X). \\
\text{concat}([X|Y], Z, [X|T]) & : - \text{ concat}(Y, Z, T).
\end{align*}
\]
\[ \text{concat}(X,Y,[2,3,4])\]?

\[ X = [] \]
\[ Y = [2, 3, 4] \]

\[ X = [2] \]
\[ Y = [3, 4] \]

\[ X = [2, 3] \]
\[ Y = [4] \]

\[ X = [2, 3, 4] \]
\[ Y = [] \]

\[ \text{reverse}(X,Y) \text{ is true if } X \text{ is a list obtained by reversing the list } Y \]

\[ \text{reverse}([],[]) \].
\[ \text{reverse}([X|Y], Z) :- \text{reverse}(Y,T), \text{concat}(T,[X],Z). \]

\[ \text{reverse}([2,3,4],X)\]?

\[ X = [4, 3, 2] \]

\[ \text{reverse}(X,[2,3,4])\]?

\[ X = [4, 3, 2] \]

GLOBAL STACK OVERFLOW
The above reverse program has been called naive reverse because it is inefficient. Note that at each step we call `concat`, to add an element at the end of the list. The computation of `concat` has to recursively descend the list to do so. This means that to reverse a list of size \( N \), we take about \( N^2 \) steps. The following is faster:

`rev2(X, Y, Z)` means \( X \) is a list, which when reversed and appended to the list \( Y \), produces the list \( Z \).

\[
\begin{align*}
\text{reverse}(X, Y) & : - \text{rev2}(X, [], Y). \\
\text{rev2}([], L, L). \\
\text{rev2}([X|Y], Z, T) & : - \text{rev2}(Y, [X|Z], T).
\end{align*}
\]

How would you have discovered this program? Probably by thinking procedurally, noting that it is faster to write the list in reverse order element by element, then to return the result. (Construct SLD trees for a few examples to see what happens when you run this program.)

This is an example of a common theme in logic programs: the use of predicates that have extra arguments that carry around information computed during the computation.

(As already noted, there are no variables to “assign” intermediate values to.)