Example:
A Datalog program defining three intentional predicates:

\[
\begin{align*}
\text{Person}(x) & \leftarrow \text{Parent}(x, y) \\
\text{Person}(y) & \leftarrow \text{Parent}(x, y) \\
\text{Grandparent}(x, z) & \leftarrow \text{Parent}(x, y), \text{Parent}(y, z) \\
\text{Ancestor}(x, z) & \leftarrow \text{Parent}(x, z) \\
\text{Ancestor}(x, z) & \leftarrow \text{Ancestor}(x, y), \text{Parent}(y, z)
\end{align*}
\]

On top of the extensional DB:

<table>
<thead>
<tr>
<th>Parent</th>
<th>P</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>juan</td>
<td>pablo</td>
<td></td>
</tr>
<tr>
<td>adam</td>
<td>cain</td>
<td></td>
</tr>
<tr>
<td>adam</td>
<td>abel</td>
<td></td>
</tr>
<tr>
<td>eve</td>
<td>cain</td>
<td></td>
</tr>
<tr>
<td>pablo</td>
<td>luis</td>
<td></td>
</tr>
</tbody>
</table>

Data propagated minimally from right to left, creating (virtual) extensions for intentional predicates.
To generate the extension for Ancestor, first apply the second last rule, moving all the data from Parent to a partial extension for Ancestor, obtaining:

\[ \text{Ancestor}' = \{ (\text{juan}, \text{pablo}), (\text{adam}, \text{cain}), (\text{adam}, \text{abel}), (\text{eve}, \text{cain}), (\text{pablo}, \text{luis}) \} \]

Now, use the last, recursive rule, evaluating the RHS, i.e. the query \( \Pi_{\text{Anc.1},\text{C}}(\text{Ancestor}' \bowtie \text{Parent}) \) (at this stage a self-join of Parent), obtaining a new partial extension for Ancestor:

\[ \text{Ancestor}'' = \text{Ancestor}' \cup \{ (\text{juan}, \text{luis}) \} \]

We use the same last rule again, a join of Ancestor'' and Parent.

We evaluate the RHS of the last rule again, but this time nothing new: we have reached a fix-point!

An the minimal model is fully computed
More on Recursion

**Example:** Descendants of \( \textit{aa} \)?

\[
\begin{align*}
Q(x) & \leftarrow \text{Ancestor}(\textit{aa}, x) & (1) \\
\text{Ancestor}(x, y) & \leftarrow \text{Parent}(x, y) & (2) \\
\text{Ancestor}(x, y) & \leftarrow \text{Parent}(x, z), \text{Ancestor}(z, y) & (3)
\end{align*}
\]

**DB \( \mathcal{D} \) of Facts:** \( \text{Parent}(a, \textit{aa}), \text{Parent}(a, \textit{ab}), \text{Parent}(\textit{aa}, \textit{aaa}), \text{Parent}(\textit{aa}, \textit{aab}), \text{Parent}(\textit{aaa}, \textit{aaaa}), \text{Parent}(c, \textit{ca}) \)

Like having a selection/projection query (the first rule) on top of a recursively defined view; in its turn defined on top of a RDB

\[
\begin{align*}
x & \quad \text{Ancestor} \\
\text{Parent} & \\
z & \quad \text{Ancestor} \\
y &
\end{align*}
\]
**Dependency Graph:**

```
Q → [Ancestor] Parent
```

(Ancestor is recursive, an iteration)

**Computation:**

1. Initialize Ancestor and Q (query answer predicate) as empty:
   
   $$Ancestor = \emptyset \quad Q = \emptyset$$

2. View Ancestor needs to be computed

   First $$Ancestor = \emptyset$$

   Apply rule (2) once, obtaining by forward propagation:

   $$Ancestor = \{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca)\}$$

   This is a partial computation of the view
3. Apply (3) with the tuples obtained in the previous step as input for the right-hand side, and propagate to the head

That is, perform the join \( \text{Parent} \times \text{"Ancestor"} \), with "Ancestor" only a partial version of the (final) \( \text{Ancestor} \)

Newly generated tuples for \( \text{Ancestor} \): \((a, \text{aaa}), (a, \text{aab}), (aa, \text{aaaa})\)

New state: \( \text{Ancestor} = \{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaaa), (c, ca), (a, aaa), (a, aab), (aa, aaaaa)\}\)

4. Since new tuples were generated wrt 1., apply rule (3) again, with the partial extension for \( \text{Ancestor} \) as input, from righ to left (forwards)

   Newly generated tuples for \( \text{Ancestor} \):
   \((a, \text{aaa}), (a, \text{aab}), (aa, \text{aaaa}), (a, \text{aaaa})\)

The underlined tuples were recomputed!

New state: \( \text{Ancestor} = \{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaaa), (c, ca), (a, aaa), (a, aab), (aa, aaaaa), (a, aaaaa)\}\)
5. Since new tuples were generated, apply rule (3) once more
   Generated tuples for \textit{Ancestor}:
   \((a, aaa), (a, aab), (aa, aaaa), (a, aaaa)\)

6. No new tuples were obtained \textit{(redundant recomputation!)};
   same state
   Block for \textit{Ancestor} is completely computed

7. Now compute the extension of \(Q\) applying its defining rule (a
   selection followed by a projection)
   Generated tuples: \(Q = \{aaa, aaaa, aab\}\)
   Same result is obtained by computing the minimal model for the
   program consisting of rules (2)-(3) plus \(D\), and posing the top
   query (1) to the minimal model
   The same minimal model can be used for different top queries
On the Minimal Model

- The minimal model can be computed in polynomial time in the size of the extensional DB (EDB) i.e. in data complexity (i.e. varying the EDB, but keeping the program fixed)

- Like creating a new database that extends EDB and querying it as usual

- Alternatively, pose queries to the new virtual database, without materializing first and posing a usual query next
• One can pose the query directly on top of the program that defines the extension, as an extra, top layer
  Propagating upwards data to answer the top query
  But maybe too much data, that is not needed for the top query

• In the example above, the full TC of Parent is computed, and moved upwards, to the top query level
  Being only interested in descendants of aa, the top selection discards too many computed tuples

• All computations before last step (7.) were done without considering the parameter aa in the query
  Many tuples were carried to the upper level and then filtered out: too much useless computation
  This part of the process can be optimized
• There is not only computation of “irrelevant” tuples
Also recomputation of tuples

• The method applied above for the computation of the minimal model is the “naive method”
This other part of the process can also be optimized

• With bottom-up evaluation we get all the answers at once

• There are many query optimization methods
Recursion in SQL3

Datalog is built-in in the newer standards of SQL
SQL3 allows to define recursive views, possibly with stratified negation
In a RDBMS using SQL3, we want to answer queries like

Examples:

1. Relation $ParentOf(parent,child)$
   Query: Find all ancestors of Mary

2. “Explosion of Parts”
   Relations: $PartOf(part\#,subpart\#)$, $Cost(part\#,price)$
   Query: What is the total cost of part #123?
   Each part consists of subparts, each subpart of subsubparts, etc.
Example: Ancestors of Mary given the table ParentOf(parent,child)?

WITH RECURSIVE Ancestor(anc,desc) AS
    ((SELECT parent AS anc, child AS desc
     FROM ParentOf)
       UNION
    (SELECT Ancestor.anc, ParentOf.child AS desc
     FROM Ancestor, ParentOf
     WHERE Ancestor.desc = ParentOf.parent))
SELECT anc
FROM Ancestor
WHERE desc = "Mary";

Notice that the definition contains the base case, and the properly recursive case.
The union corresponds to the two rules used to define the new predicate.
Example:  Total cost of part #123 from
  PartOf(part#,subpart#) and Cost(part#,price)?

WITH  RECURSIVE AllParts AS
( (SELECT * FROM PartOf)
UNION
( SELECT  A1.part#, A2.part#
FROM    AllParts A1, AllParts A2
WHERE   A1.subpart# = A2.part# )
)
SELECT sum(Cost.price)
FROM    AllParts, Cost
WHERE   AllParts.part# = 123
        AND AllParts.subpart# = Cost.part#
Several extensions of Datalog

E.g. aggregation functions

\[
Ans(x, sum(z)) \leftarrow R(x, y, z)
\]

Addition with group-by
Top-Down Query Evaluation

Uses the resolution deductive rule

Program $\Pi$:

\[
\begin{align*}
\text{burglary}. & \quad \text{hearsAlarm}(\text{mary}). & \quad \text{earthquake}. & \quad \text{hearsAlarm}(\text{john}). \\
\text{alarm} & \leftarrow \text{earthquake}. & \quad (1) \\
\text{alarm} & \leftarrow \text{burglary}. & \quad (2) \\
\text{calls}(X) & \leftarrow \text{alarm}, \text{hearsAlarm}(X). & \quad (3) \\
\text{call} & \leftarrow \text{calls}(X). & \quad (4)
\end{align*}
\]

Positive Datalog

Two ways to evaluate queries:

1. Build the minimal model and query it as a RDB

Bottom-up approach (typical of Datalog)

Minimal Model $\mathcal{M}$ contains:

- Facts: $\text{burglary}, \text{hearsAlarm}(\text{mary}), \text{earthquake}, \text{hearsAlarm}(\text{john})$
- Derived atoms: $\text{alarm}, \text{calls}(\text{mary}), \text{calls}(\text{john}), \text{call}$

Query: $\neg \text{call}?$ \quad Yes! \quad (by querying $\mathcal{M}$)
2. Query-dependent methodology based on resolution

Top-down approach (typical of Prolog)

• Query: *call?* So, we try to prove/deduce atom *call*

• We add it in negated form to the program: ¬ *call*

It will be a proof by contradiction via resolution

A contradiction is a clause that is always false: the empty clause □ (no literals)

• Good enough: Remember equiconsistency result from Chapter 1

Negative literal, written in clausal rule form: ← *call*

• Resolution with (4):

  ← ¬ *call*
  ¬ *call* ← *calls(X)*
  *call* ← *calls(X)*

  ← *calls(X)*  new goal

• Etc. until reaching □
Prolog follows leftmost path in depth (in red) with backtracking

Search guided by the query (contrary to bottom-up)

One answer at a time (slower than bottom-up)

\[ \text{\texttt{\):- the same as \texttt{←} (Prolog notation)}} \]

\[ \text{\texttt{\):- \texttt{call} equivalent to \texttt{¬call}}] \]

\[ \text{\texttt{\):- \texttt{calls(x) equivalent to \texttt{¬∃xcall(x)}}} \]

Above: a successful (resolution-based) refutation tree
• Above *Yes!* means success, i.e. the empty clause $\square$ was reached

• At the root we have the negation of what we want to prove, in the form: $\leftarrow \text{call}$

• The unifications that lead to success are witnesses for the implicit existential variables
  Those values provide the query answers

• Bottom-up vs. Top-down query evaluation?
  Model-based vs. procedural semantics?
  Being true in intended model vs. existence of refutation tree?

• For (positive) Datalog programs $\Pi$ and conjunctive queries $Q$, both query evaluation methods are equivalent
  They return exactly the same answers