10.3 Recursive Programming in Datalog

While relational algebra can express many useful operations on relations, there are some computations that cannot be written as an expression of relational algebra. A common kind of operation on data that we cannot express in relational algebra involves an infinite, recursively defined sequence of similar expressions.

Example 10.1: Often, a successful movie is followed by a sequel; if the sequel does well, then the sequel has a sequel, and so on. Thus, a movie may be ancestral to a long sequence of other movies. Suppose we have a relation `SequelOf(movie, sequel)` containing pairs consisting of a movie and its immediate sequel. Examples of tuples in this relation are:

movie	*sequel*
`Naked Gun`	`Naked Gun 21/2`
`Naked Gun 21/2`	`Naked Gun 331/3`

We might also have a more general notion of a *follow-on* to a movie, which is a sequel, a sequel of a sequel, and so on. In the relation above, *Naked Gun 331/3* is a follow-on to *Naked Gun*, but not a sequel in the strict sense we are using the term "sequel" here. It saves space if we store only the immediate sequels in the relation and construct the follow-ons if we need them. In the above example, we store only one fewer pair, but for the five *Rocky* movies we store six fewer pairs, and for the 18 *Friday the 13th* movies we store 136 fewer pairs.

However, it is not immediately obvious how we construct the relation of follow-ons from the relation `SequelOf`. We can construct the sequels of sequels by joining `SequelOf` with itself once. An example of such an expression in relational algebra, using renaming so that the join becomes a natural join, is:

$$\pi_{first,third}\big(\rho_{R(first,second)}(\texttt{SequelOf}) \bowtie \rho_{S(second,third)}(\texttt{SequelOf})\big)$$

In this expression, `SequelOf` is renamed twice, once so its attributes are called `first` and `second`, and again so its attributes are called `second` and `third`. Thus, the natural join asks for tuples (m_1, m_2) and (m_3, m_4) in `SequelOf` such that $m_2 = m_3$. We then produce the pair (m_1, m_4). Note that m_4 is the sequel of the sequel of m_1.

Similarly, we could join three copies of `SequelOf` to get the sequels of sequels of sequels (e.g., *Rocky* and *Rocky IV*). We could in fact produce the ith sequels for any fixed value of i by joining `SequelOf` with itself $i-1$ times. We could then take the union of `SequelOf` and a finite sequence of these joins to get all the sequels up to some fixed limit.

What we cannot do in relational algebra is ask for the "infinite union" of the infinite sequence of expressions that give the ith sequels for $i = 1, 2, \ldots$. Note that relational algebra's union allows us only to take the union of two relations, not an infinite number. By applying the union operator any finite number of times in an algebraic expression, we can take the union of any finite number of relations, but we cannot take the union of an unlimited number of relations in an algebraic expression. □

10.3.1 Recursive Rules

By using an IDB predicate both in the head and the body of rules, we can express an infinite union in Datalog. We shall first see some examples of how to express recursions in Datalog. In Section 10.3.2 we shall examine the *least fixedpoint* computation of the relations for the IDB predicates of these rules. A new approach to rule-evaluation is needed for recursive rules, since the straightforward rule-evaluation approach of Section ?? assumes all the predicates in the body of rules have fixed relations.

Example 10.2: We can define the IDB relation `FollowOn` by the following two Datalog rules:

1. `FollowOn(x,y)` $\leftarrow$ `SequelOf(x,y)`
2. `FollowOn(x,y)` $\leftarrow$ `SequelOf(x,z) AND FollowOn(z,y)`

The first rule is the basis; it tells us that every sequel is a follow-on. The second rule says that every follow-on of a sequel of movie x is also a follow-on of x. More precisely: if z is a sequel of x, and we have found that y is a follow-on of z, then y is a follow-on of x. □

10.3.2 Evaluating Recursive Datalog Rules

To evaluate the IDB predicates of recursive Datalog rules, we follow the principle that we never want to conclude that a tuple is in an IDB relation unless we are forced to do so by applying the rules as in Section ??. Thus, we:

1. Begin by assuming all IDB predicates have empty relations.

2. Perform a number of *rounds*, in which progressively larger relations are constructed for the IDB predicates. In the bodies of the rules, use the IDB relations constructed on the previous round. Apply the rules to get new estimates for all the IDB predicates.

3. If the rules are safe, no IDB tuple can have a component value that does not also appear in some EDB relation. Thus, there are a finite number of possible tuples for all IDB relations, and eventually there will be a round on which no new tuples are added to any IDB relation. At this point, we can terminate our computation with the answer; no new IDB tuples will ever be constructed.

This set of IDB tuples is called the *least fixedpoint* of the rules.

Example 10.3: Let us show the computation of the least fixedpoint for relation `FollowOn` when the relation `SequelOf` consists of the following three tuples:

movie	*sequel*
`Rocky`	`Rocky II`
`Rocky II`	`Rocky III`
`Rocky III`	`Rocky IV`

At the first round of computation, `FollowOn` is assumed empty. Thus, rule (2) cannot yield any `FollowOn` tuples. However, rule (1) says that every `SequelOf` tuple is a `FollowOn` tuple. Thus, after the first round, the value of `FollowOn` is identical to the `SequelOf` relation above. The situation after round 1 is shown in Fig. 10.1(a).

In the second round, we use the relation from Fig. 10.1(a) as `FollowOn` and apply the two rules to this relation and the given `SequelOf` relation. The first rule gives us the three tuples that we already have, and in fact it is easy to see that rule (1) will never yield any tuples for `FollowOn` other than these three. For rule (2), we look for a tuple from `SequelOf` whose second component equals the first component of a tuple from `FollowOn`.

Thus, we can take the tuple (`Rocky`, `Rocky II`) from `SequelOf` and pair it with the tuple (`Rocky II`, `Rocky III`) from `FollowOn` to get the new tuple (`Rocky`, `Rocky III`) for `FollowOn`. Similarly, we can take the tuple

(`Rocky II`, `Rocky III`)

from `SequelOf` and tuple (`Rocky III`, `Rocky IV`) from `FollowOn` to get new tuple (`Rocky II`, `Rocky IV`) for `FollowOn`. However, no other pairs of tuples from `SequelOf` and `FollowOn` join. Thus, after the second round, `FollowOn` has the five tuples shown in Fig. 10.1(b). Intuitively, just as Fig. 10.1(a) contained only those follow-on facts that are based on a single sequel, Fig. 10.1(b) contains those follow-on facts based on one or two sequels.

In the third round, we use the relation from Fig. 10.1(b) for `FollowOn` and again evaluate the body of rule (2). We get all the tuples we already had, of course, and one more tuple. When we join the tuple (`Rocky`, `Rocky II`) from `SequelOf` with the tuple (`Rocky II`, `Rocky IV`) from the current value of `FollowOn`, we get the new tuple (`Rocky`, `Rocky IV`). Thus, after round 3, the value of `FollowOn` is as shown in Fig. 10.1(c).

When we proceed to round 4, we get no new tuples, so we stop. The true relation `FollowOn` is as shown in Fig. 10.1(c). □

There is an important trick that simplifies all recursive Datalog evaluations, such as the one above:

- At any round, the only new tuples added to any IDB relation will come from applications of rules in which at least one IDB subgoal is matched to a tuple that was added to its relation at the previous round.

x	y
`Rocky`	`Rocky II`
`Rocky II`	`Rocky III`
`Rocky III`	`Rocky IV`

(a) After round 1

x	y
`Rocky`	`Rocky II`
`Rocky II`	`Rocky III`
`Rocky III`	`Rocky IV`
`Rocky`	`Rocky III`
`Rocky II`	`Rocky IV`

(b) After round 2

x	y
`Rocky`	`Rocky II`
`Rocky II`	`Rocky III`
`Rocky III`	`Rocky IV`
`Rocky`	`Rocky III`
`Rocky II`	`Rocky IV`
`Rocky`	`Rocky IV`

(c) After round 3 and subsequently

Figure 10.1: Recursive computation of relation `FollowOn`

The justification for this rule is that should all subgoals be matched to "old" tuples, the tuple of the head would already have been added on the previous round. The next two examples illustrate this strategy and also show us more complex examples of recursion.

Example 10.4 : Many examples of the use of recursion can be found in a study of paths in a graph. Figure 10.2 shows a graph representing some flights of two hypothetical airlines — *Untried Airlines* (UA), and *Arcane Airlines* (AA) — among the cities San Francisco, Denver, Dallas, Chicago, and New York.

We may imagine that the flights are represented by an EDB relation:

```
Flights(airline, from, to, departs, arrives)
```

The tuples in this relation for the data of Fig. 10.2 are shown in Fig. 10.3.

Other Forms of Recursion

In Example 10.2 we used a *right-recursive* form for the recursion, where the use of the recursive relation `FollowOn` appears after the EDB relation `SequelOf`. We could also write similar *left-recursive* rules by putting the recursive relation first. These rules are:

1. `FollowOn(x,y)` ← `SequelOf(x,y)`
2. `FollowOn(x,y)` ← `FollowOn(x,z) AND SequelOf(z,y)`

Informally, y is a follow-on of x if it is either a sequel of x or a sequel of a follow-on of x.

We could even use the recursive relation twice, as in the *nonlinear* recursion:

1. `FollowOn(x,y)` ← `SequelOf(x,y)`
2. `FollowOn(x,y)` ← `FollowOn(x,z) AND FollowOn(z,y)`

Informally, y is a follow-on of x if it is either a sequel of x or a follow-on of a follow-on of x. All three of these forms give the same value for relation `FollowOn`: the set of pairs (x, y) such that y is a sequel of a sequel of $\cdots$ (some number of times) of x.

The simplest recursive question we can ask is "For what pairs of cities (x, y) is it possible to get from city x to city y by taking one or more flights?" The following two rules describe a relation `Reaches(x,y)` that contains exactly these pairs of cities.

1. `Reaches(x,y)` ← `Flights(a,x,y,d,r)`
2. `Reaches(x,y)` ← `Reaches(x,z) AND Reaches(z,y)`

The first rule says that `Reaches` contains those pairs of cities for which there is a direct flight from the first to the second; the airline a, departure time d, and arrival time r are arbitrary in this rule. The second rule says that if you can reach from city x to city z and you can reach from z to y, then you can reach from x to y. Notice that we have used the nonlinear form of recursion here, as was described in the box on "Other Forms of Recursion." This form is slightly more convenient here, because another use of `Flights` in the recursive rule would involve three more variables for the unused components of `Flights`.

To evaluate the relation `Reaches`, we follow the same iterative process introduced in Example 10.3. We begin by using Rule (1) to get the following pairs in `Reaches`: (`SF`, `DEN`), (`SF`, `DAL`), (`DEN`, `CHI`), (`DEN`, `DAL`), (`DAL`, `CHI`), (`DAL`, `NY`), and (`CHI`, `NY`). These are the seven pairs represented by arcs in Fig. 10.2.

In the next round, we apply the recursive Rule (2) to put together pairs of arcs such that the head of one is the tail of the next. That gives us the

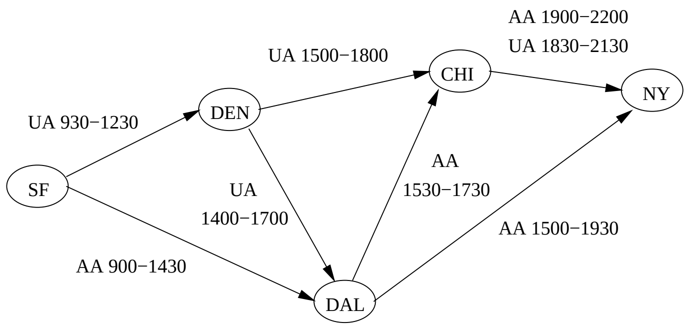

Figure 10.2: A map of some airline flights

airline	*from*	*to*	*departs*	*arrives*
UA	SF	DEN	930	1230
AA	SF	DAL	900	1430
UA	DEN	CHI	1500	1800
UA	DEN	DAL	1400	1700
AA	DAL	CHI	1530	1730
AA	DAL	NY	1500	1930
AA	CHI	NY	1900	2200
UA	CHI	NY	1830	2130

Figure 10.3: Tuples in the relation `Flights`

additional pairs (`SF`, `CHI`), (`DEN`, `NY`), and (`SF`, `NY`). The third round combines all one- and two-arc pairs together to form paths of length up to four arcs. In this particular diagram, we get no new pairs. The relation `Reaches` thus consists of the ten pairs (x, y) such that y is reachable from x in the diagram of Fig. 10.2. Because of the way we drew the diagram, these pairs happen to be exactly those (x, y) such that y is to the right of x in Fig 10.2. □

Example 10.5: A more complicated definition of when two flights can be combined into a longer sequence of flights is to require that the second leaves an airport at least an hour after the first arrives at that airport. Now, we use an IDB predicate, which we shall call `Connects(x,y,d,r)`, that says we can take one or more flights, starting at city x at time d and arriving at city y at time r. If there are any connections, then there is at least an hour to make the connection.

The rules for `Connects` are:[1]

[1] These rules only work on the assumption that there are no connections spanning midnight.

```
1.  Connects(x,y,d,r)  ←  Flights(a,x,y,d,r)
2.  Connects(x,y,d,r)  ←  Connects(x,z,d,t1) AND
                           Connects(z,y,t2,r) AND
                           t1 <= t2 - 100
```

In the first round, rule (1) gives us the eight `Connects` facts shown above the first line in Fig. 10.4 (the line is not part of the relation). Each corresponds to one of the flights indicated in the diagram of Fig. 10.2; note that one of the seven arcs of that figure represents two flights at different times.

We now try to combine these tuples using Rule (2). For example, the second and fifth of these tuples combine to give the tuple (`SF`, `CHI`, 900, 1730). However, the second and sixth tuples do not combine because the arrival time in Dallas is 1430, and the departure time from Dallas, 1500, is only half an hour later. The `Connects` relation after the second round consists of all those tuples above the first or second line in Fig. 10.4. Above the top line are the original tuples from round 1, and the six tuples added on round 2 are shown between the first and second lines.

x	y	d	r
SF	DEN	930	1230
SF	DAL	900	1430
DEN	CHI	1500	1800
DEN	DAL	1400	1700
DAL	CHI	1530	1730
DAL	NY	1500	1930
CHI	NY	1900	2200
CHI	NY	1830	2130
SF	CHI	900	1730
SF	CHI	930	1800
SF	DAL	930	1700
DEN	NY	1500	2200
DAL	NY	1530	2130
DAL	NY	1530	2200
SF	NY	900	2130
SF	NY	900	2200
SF	NY	930	2200

Figure 10.4: Relation `Connects` after third round

In the third round, we must in principle consider all pairs of tuples above one of the two lines in Fig. 10.4 as candidates for the two `Connects` tuples in the body of rule (2). However, if both tuples are above the first line, then they would have been considered during round 2 and therefore will not yield a `Connects` tuple we have not seen before. The only way to get a new tuple is if at least one of the two `Connects` tuple used in the body of rule (2) were added at the previous round; i.e., it is between the lines in Fig. 10.4.

The third round only gives us three new tuples. These are shown at the bottom of Fig. 10.4. There are no new tuples in the fourth round, so our computation is complete. Thus, the entire relation `Connects` is Fig. 10.4. □

10.3.3 Negation in Recursive Rules

Sometimes it is necessary to use negation in rules that also involve recursion. There is a safe way and an unsafe way to mix recursion and negation. Generally, it is considered appropriate to use negation only in situations where the negation does not appear inside the fixedpoint operation. To see the difference, we shall consider two examples of recursion and negation, one appropriate and the other paradoxical. We shall see that only "stratified" negation is useful when there is recursion; the term "stratified" will be defined precisely after the examples.

Example 10.6: Suppose we want to find those pairs of cities (x, y) in the map of Fig. 10.2 such that UA flies from x to y (perhaps through several other cities), but AA does not. We can recursively define a predicate `UAreaches` as we defined `Reaches` in Example 10.4, but restricting ourselves only to UA flights, as follows:

1. `UAreaches(x,y)` ← `Flights(UA,x,y,d,r)`
2. `UAreaches(x,y)` ← `UAreaches(x,z) AND UAreaches(z,y)`

Similarly, we can recursively define the predicate `AAreaches` to be those pairs of cities (x, y) such that one can travel from x to y using only AA flights, by:

1. `AAreaches(x,y)` ← `Flights(AA,x,y,d,r)`
2. `AAreaches(x,y)` ← `AAreaches(x,z) AND AAreaches(z,y)`

Now, it is a simple matter to compute the `UAonly` predicate consisting of those pairs of cities (x, y) such that one can get from x to y on UA flights but not on AA flights, with the nonrecursive rule:

```
UAonly(x,y) ← UAreaches(x,y) AND NOT AAreaches(x,y)
```

This rule computes the set difference of `UAreaches` and `AAreaches`.

For the data of Fig. 10.2, `UAreaches` is seen to consist of the following pairs: (`SF`, `DEN`), (`SF`, `DAL`), (`SF`, `CHI`), (`SF`, `NY`), (`DEN`, `DAL`), (`DEN`, `CHI`), (`DEN`, `NY`), and (`CHI`, `NY`). This set is computed by the iterative fixedpoint process outlined in Section 10.3.2. Similarly, we can compute the value of `AAreaches` for this data; it is: (`SF`, `DAL`), (`SF`, `CHI`), (`SF`, `NY`), (`DAL`, `CHI`), (`DAL`, `NY`), and (`CHI`, `NY`). When we take the difference of these sets of pairs we get: (`SF`, `DEN`), (`DEN`, `DAL`), (`DEN`, `CHI`), and (`DEN`, `NY`). This set of four pairs is the relation `UAonly`. □

Example 10.7: Now, let us consider an abstract example where things don't work as well. Suppose we have a single EDB predicate R. This predicate is unary (one-argument), and it has a single tuple, (0). There are two IDB predicates, P and Q, also unary. They are defined by the two rules

1. `P(x)` $\leftarrow$ `R(x) AND NOT Q(x)`
2. `Q(x)` $\leftarrow$ `R(x) AND NOT P(x)`

Informally, the two rules tell us that an element x in R is either in P or in Q but not both. Notice that P and Q are defined recursively in terms of each other.

When we defined what recursive rules meant in Section 10.3.2, we said we want the least fixedpoint, that is, the smallest IDB relations that contain all tuples that the rules require us to allow. Rule (1), since it is the only rule for P, says that as relations, $P = R - Q$, and rule (2) likewise says that $Q = R - P$. Since R contains only the tuple (0), we know that only (0) can be in either P or Q. But where is (0)? It cannot be in neither, since then the equations are not satisfied; for instance $P = R - Q$ would imply that $\emptyset = \{(0)\} - \emptyset$, which is false.

If we let $P = \{(0)\}$ while $Q = \emptyset$, then we do get a solution to both equations. $P = R - Q$ becomes $\{(0)\} = \{(0)\} - \emptyset$, which is true, and $Q = R - P$ becomes $\emptyset = \{(0)\} - \{(0)\}$, which is also true.

However, we can also let $P = \emptyset$ and $Q = \{(0)\}$. This choice too satisfies both rules. We thus have two solutions:

a)	$P = \{(0)\}$	$Q = \emptyset$
b)	$P = \emptyset$	$Q = \{(0)\}$

Both are minimal, in the sense that if we throw any tuple out of any relation, the resulting relations no longer satisfy the rules. We cannot, therefore, decide between the two least fixedpoints (a) and (b), so we cannot answer a simple question such as "Is $P(0)$ true?" $\Box$

In Example 10.7, we saw that our idea of defining the meaning of recursive rules by finding the least fixedpoint no longer works when recursion and negation are tangled up too intimately. There can be more than one least fixedpoint, and these fixedpoints can contradict each other. It would be good if some other approach to defining the meaning of recursive negation would work better, but unfortunately, there is no general agreement about what such rules should mean.

Thus, it is conventional to restrict ourselves to recursions in which negation is *stratified*. For instance, the SQL-99 standard for recursion discussed in Section **??** makes this restriction. As we shall see, when negation is stratified there is an algorithm to compute one particular least fixedpoint (perhaps out of many such fixedpoints) that matches our intuition about what the rules mean. We define the property of being stratified as follows.

1. Draw a graph whose nodes correspond to the IDB predicates.

2. Draw an arc from node A to node B if a rule with predicate A in the head has a negated subgoal with predicate B. Label this arc with a $-$ sign to indicate it is a *negative* arc.

3. Draw an arc from node A to node B if a rule with head predicate A has a non-negated subgoal with predicate B. This arc does not have a minus-sign as label.

If this graph has a cycle containing one or more negative arcs, then the recursion is not stratified. Otherwise, the recursion is stratified. We can group the IDB predicates of a stratified graph into *strata*. The stratum of a predicate A is the largest number of negative arcs on a path beginning from A.

If the recursion is stratified, then we may evaluate the IDB predicates in the order of their strata, lowest first. This strategy produces one of the least fixedpoints of the rules. More importantly, computing the IDB predicates in the order implied by their strata appears always to make sense and give us the "right" fixedpoint. In contrast, as we have seen in Example 10.7, unstratified recursions may leave us with no "right" fixedpoint at all, even if there are many to choose from.

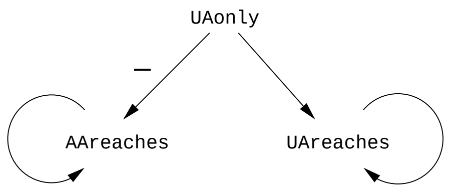

Figure 10.5: Graph constructed from a stratified recursion

Example 10.8: The graph for the predicates of Example 10.6 is shown in Fig. 10.5. `AAreaches` and `UAreaches` are in stratum 0, because none of the paths beginning at their nodes involves a negative arc. `UAonly` has stratum 1, because there are paths with one negative arc leading from that node, but no paths with more than one negative arc. Thus, we must completely evaluate `AAreaches` and `UAreaches` before we start evaluating `UAonly`.

Compare the situation when we construct the graph for the IDB predicates of Example 10.7. This graph is shown in Fig. 10.6. Since rule (1) has head P with negated subgoal Q, there is a negative arc from P to Q. Since rule (2) has head Q with negated subgoal P, there is also a negative arc in the opposite direction. There is thus a negative cycle, and the rules are not stratified. □

$$\begin{tikzcd} P \arrow[r, bend left, "-"] & Q \arrow[l, bend left, "-"] \end{tikzcd}$$

Figure 10.6: Graph constructed from an unstratified recursion

10.3.4 Exercises for Section 10.3

Exercise 10.3.1: If we add or delete arcs to the diagram of Fig. 10.2, we may change the value of the relation `Reaches` of Example 10.4, the relation `Connects` of Example 10.5, or the relations `UAreaches` and `AAreaches` of Example 10.6. Give the new values of these relations if we:

* a) Add an arc from `CHI` to `SF` labeled AA, 1900-2100.

b) Add an arc from `NY` to `DEN` labeled UA, 900-1100.

c) Add both arcs from (a) and (b).

d) Delete the arc from `DEN` to `DAL`.

Exercise 10.3.2: Write Datalog rules (using stratified negation, if negation is necessary) to describe the following modifications to the notion of "follow-on" from Example 10.1. You may use EDB relation `SequelOf` and the IDB relation `FollowOn` defined in Example 10.2.

* a) `P(x,y)` meaning that movie y is a follow-on to movie x, but not a sequel of x (as defined by the EDB relation `SequelOf`).

b) `Q(x,y)` meaning that y is a follow-on of x, but neither a sequel nor a sequel of a sequel.

! c) `R(x)` meaning that movie x has at least two follow-ons. Note that both could be sequels, rather than one being a sequel and the other a sequel of a sequel.

!! d) `S(x,y)`, meaning that y is a follow-on of x but y has at most one follow-on.

Exercise 10.3.3: ODL classes and their relationships can be described by a relation `Rel(class, rclass, mult)`. Here, `mult` gives the multiplicity of a relationship, either `multi` for a multivalued relationship, or `single` for a single-valued relationship. The first two attributes are the related classes; the relationship goes from `class` to `rclass` (related class). For example, the relation `Rel` representing the three ODL classes of our running movie example from Fig. ?? is shown in Fig. 10.7.

We can also see this data as a graph, in which the nodes are classes and the arcs go from a class to a related class, with label `multi` or `single`, as appropriate. Figure 10.8 illustrates this graph for the data of Fig. 10.7.

class	*rclass*	*mult*
`Star`	`Movie`	`multi`
`Movie`	`Star`	`multi`
`Movie`	`Studio`	`single`
`Studio`	`Movie`	`multi`

Figure 10.7: Representing ODL relationships by relational data

$$\begin{tikzcd}
\text{Star} \arrow[r, bend left, "\text{multi}"] & \text{Movie} \arrow[l, bend left, "\text{multi}"] \arrow[r, bend left, "\text{single}"] & \text{Studio} \arrow[l, bend left, "\text{multi}"]
\end{tikzcd}$$

Figure 10.8: Representing relationships by a graph

For each of the following, write Datalog rules, using stratified negation if negation is necessary, to express the described predicate(s). You may use `Rel` as an EDB relation. Show the result of evaluating your rules, round-by-round, on the data from Fig. 10.7.

a) Predicate `P(class, eclass)`, meaning that there is a path[2] in the graph of classes that goes from `class` to `eclass`. The latter class can be thought of as "embedded" in `class`, since it is in a sense part of a part of an $\cdots$ object of the first class.

*! b) Predicates `S(class, eclass)` and `M(class, eclass)`. The first means that there is a "single-valued embedding" of `eclass` in `class`, that is, a path from `class` to `eclass` along which every arc is labeled `single`. The second, M, means that there is a "multivalued embedding" of `eclass` in `class`, i.e., a path from `class` to `eclass` with at least one arc labeled `multi`.

c) Predicate `Q(class, eclass)` that says there is a path from `class` to `eclass` but no single-valued path. You may use IDB predicates defined previously in this exercise.

[2]We shall not consider empty paths to be "paths" in this exercise.