# Exercise 1

## Datalog and Network Reliability

Program Analysis for System Security and Reliability 2018
ETH Zurich

March 6, 2018

**LogicBlox** LogicBlox is a popular Datalog engine. You can write and query Datalog programs via its web interface, which is publicly available at

`https://repl.logicblox.com`

You will use this interface for all tasks in this exercise.

To illustrate LogicBlox, let's encode the "family tree" example that you have seen in the lecture. When writing Datalog programs in LogicBlox, one needs to explicitly declare the argument types of input relations. The following rule declares that the argument types of the relation `parent` are `String`:

```
addblock 'parent(X, Y) -> string(X), string(Y).'
```

You can copy and paste the above block directly into the LogicBlox web interface.

Next, we can add two tuples to the `parent` relation:

```
exec '+parent("bob", "alice").
      +parent("dave", "bob").'
```

We can then define the ancestor relation, called `anc`, as the transitive closure of the `parent` relation:

```
addblock 'anc(X, Y) <- parent(X, Y).
          anc(X, Y) <- parent(X, Z), anc(Z, Y). '
```

Finally, we can query the fixed point of the Datalog program. For instance, to print which tuples are contained in `anc` relation, you can run the command `print anc`.

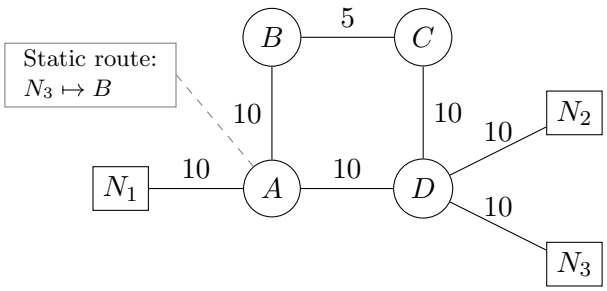


Figure 1: Topology of the network. Circles represent routers, and squares represent neighboring networks. All links are bidirectional. Each link is associated with a weight (a positive integer). Router $A$ has a static route that specifies that packets destined to network $N_3$ must be forwarded to router $B$.

For more detail on LogicBlox and the RPEL interface you can follow this tutorial:
`https://developer.logicblox.com/content/docs4/tutorial/repl/section/split.html`.

**Problem 1.** In this task, you will use stratified Datalog to verifying a network configuration. We consider a network that consists of 4 routers—$A$, $B$, $C$, and $D$. Our network carries traffic for three neighbor networks—$N_1$, $N_2$, and $N_3$. The network topology is shown in Figure 1. Each links is associated with a weight (shown in Figure 1).

Each router computes its own forwarding entries according to its local configuration, by running the OSPF protocol and composing the computed OSPF routes with any static routes defined in the configuration. OSPF is an intra-domain routing protocol used by routers to compute forwarding entries directing traffic to any internal destinations along the shortest-path based on configurable link weights. Each router composes the routes computed OSPF with those statically defined in the configuration based on the local preference configured in the router's configuration. For our example network, the routers assign the local preference 10 to static routes and 5 to OSPF (i.e., routers would select any static route, if defined, over the computed OSPF routs). Router $A$ has a static route that specifies that packets destined to network $N_3$ must be forwarded to router $B$. The global forwarding plane of the network can be captured with a relation `Fwd(Router, Network, NextHop)`. For example, the tuple `Fwd(A, N2, D)` captures that router $A$ forwards packets destined to network $N_2$ to router $D$.

**Task 1: Encode the network configuration as a Datalog input** Define several relation that captures the network configuration. Relevant pieces of information that you must capture include: (1) network links and their costs, (2) the local preference of each router, and (3) static routes defined in the router configurations. Define the necessary relations using the LogicBLox syntax.

**Solution:**

```
network(X) -> string(X).
router(X) -> string(X).
protocol(X) -> string(X).
connected(X, Y, Z) -> string(X), string(Y), int(Z).
static(router, Network, NextHop) -> string(router), string(Network), string(
    NextHop).
localPref(router, Protocol, PrefValue) -> string(router), string(Protocol),
    int(PrefValue).
router("A").
router("B").
router("C").
router("D").
network("N1").
network("N2").
network("N3").
protocol("OSPF").
protocol("static").
connected("N1", "A", 10).
connected("N2", "D", 10).
connected("N3", "D", 10).
connected("A", "B", 10).
connected("B", "C", 5).
connected("C", "D", 10).
connected("D", "A", 10).
static("A", "N3", "B").
localPref("A", "static", 10).
localPref("B", "static", 10).
localPref("C", "static", 10).
localPref("D", "static", 10).
localPref("A", "OSPF", 5).
localPref("B", "OSPF", 5).
localPref("C", "OSPF", 5).
localPref("D", "OSPF", 5).
```

**Task 2: Encode network behavior** Write a Datalog program that encodes how routers compute their forwarding entries. You must first specify how routers compute their OSPF routes (based on shortest path) and then specify how OSPF routes are composed with any statically defined entries (based on the router's local preference). Your Datalog program must define the global forwarding plane of the network, i.e. the relation `Fwd(Router, Network, NextHop)`.

```
link(X, Y, Z) <- connected(X, Y, Z).
link(X, Y, Z) <- link(Y, X, Z).
OSPFroute(router, Network, Network, Cost) <- link(router, Network, Cost),
    router(router), network(Network).
OSPFroute(router, Network, NextHop, Cost) <- router(router), router(NextHop),
     link(router, NextHop, Cost1), OSPFroute(NextHop, Network, _, Cost2),
    Cost = Cost1 + Cost2, Cost < 50.
nonMinNextHop(router, Network, NextHop, Cost1) <- OSPFroute(router, Network,
    NextHop, Cost1), OSPFroute(router, Network, _, Cost2), Cost1 > Cost2.
```

```
bestOSPFroute(router, Network, NextHop) <- OSPFroute(router, Network, NextHop
    , Cost), !nonMinNextHop(router, Network, NextHop, Cost).
route(router, Network, Protocol, NextHop) <- bestOSPFroute(router, Network,
    NextHop), Protocol="ospf".
route(router, Network, Protocol, NextHop) <- static(router, Network, NextHop)
    , Protocol="static".
nonPrefroute(router, Network, Protocol1) <- route(router, Network, Protocol1,
     _), route(router, Network, Protocol2, _), localPref(router, Protocol1,
    Pref1), localPref(router, Protocol2, Pref2), Pref1 < Pref2.
fwd(router, Network, NextHop) <- route(router, Network, Protocol, NextHop), !
    nonPrefroute(router, Network, Protocol).
```

**Task 3: Query the network's forwarding plane** Show the routers' forwarding tables.

**Solution:**

| Router $A$ | |
|---|---|
| Network | NextHop |
| N1 | N1 |
| N2 | D |
| N3 | B |

| Router $B$ | |
|---|---|
| Network | NextHop |
| N1 | A |
| N2 | C |
| N3 | C |

| Router $C$ | |
|---|---|
| Network | NextHop |
| N1 | B |
| N2 | D |
| N3 | D |

| Router $D$ | |
|---|---|
| Network | NextHop |
| N1 | A |
| N2 | N2 |
| N3 | N3 |

**Task 4: Check traffic isolation for networks $N_2$ and $N_3$** Write a Datalog query that is satisfied if and only if

$$\forall X, Y.(Fwd(X, N_2, Y) \Rightarrow (\neg Fwd(X, N_3, Y)))$$

Here, $N_2$ and $N_3$ denote the identifiers of the two networks. This property captures that the forwarding paths for networks $N_2$ and $N_3$ do not share links in the same direction.

**Solution:**

Ideally, we would like to write a Datalog rule

$$TI \leftarrow \big(\forall X, Y.(Fwd(X, N_2, Y) \Rightarrow (\neg Fwd(X, N_3, Y)))\big)$$

where the atom $TI$ is derived iff the property holds. Unfortunately, the above rule is not a valid Datalog rule since we cannot write universal quantifiers in the body of the Datalog rule. Note that the free variables that appear in the body of a Datalog rule are existentially quantified (see lecture).

We can however encode the property via several rules. The key idea is to transform the universal quantifier with an existential one in the standard way:

$$\forall X.\ \varphi(X) \Leftrightarrow \neg\exists X.\ \neg\varphi(X)$$

We can then check $\exists X.\ \neg\varphi(X)$ with a datalog rule $q \leftarrow \neg\varphi(X)$, where $q$ is derived iff $\exists X.\ \neg\varphi(X)$.

We transform the property as follows:

$$\begin{aligned}
& \forall X, Y.(Fwd(X, N_2, Y) \Rightarrow (\neg Fwd(X, N_3, Y))) && (1)\\
\Leftrightarrow\quad & \forall X, Y.(\neg Fwd(X, N_2, Y) \vee \neg Fwd(X, N_3, Y)) && (2)\\
\Leftrightarrow\quad & \neg\exists X, Y.\neg(\neg Fwd(X, N_2, Y) \vee \neg Fwd(X, N_3, Y)) && (3)\\
\Leftrightarrow\quad & \neg\exists X, Y.Fwd(X, N_2, Y) \wedge Fwd(X, N_3, Y) && (4)
\end{aligned}$$

We can now encode whether $\exists X, Y.Fwd(X, N_2, Y) \wedge Fwd(X, N_3, Y)$ holds with a Datalog rule

```
sharedLinks() <- Fwd(X, "N2", Y), Fwd(X, "N3", Y).
```

The atom `sharedLinks` is derived iff the two networks share a link from $X$ to $Y$. We can then immediately check traffic isolation with an additional rule:

```
TI(res) <- sharedLinks(), res = "traffic for N2 and N3 is NOT isolated".
TI(res) <- !sharedLinks(), res = "traffic for N2 and N3 IS isolated".
```

**Task 5: Change the network configuration** Change the router's configuration so that the traffic isolation property for $N_2$ and $N_3$ holds.

**Solution:** One way to change the configuration is to add two static routes:

```
+static("B", "N2", "A").
+static("C", "N2", "B").
```

**Problem 2. (Bonus problem)** In this task, you will use Datalog to declaratively specify a popular data-flow analysis, called reaching definitions. This analysis statically determines which assignments (a.k.a. definitions) may reach a given program label. For more information about the reaching definitions analysis and examples see `https://en.wikipedia.org/wiki/Reaching_definition`.

To analyze a program in Datalog, we must first derive facts about the program, which capture information relevant for our analysis. For the reaching definitions analysis, we need information about (1) assignments and (2) control flow. We capture these pieces of information with the following two predicates:

1. `assign(Lab, Var)`, which captures that the program makes an assignment to variable `Var` at program label `Lab`
2. `follows(Lab1, Lab2)`, which captures that program label `Lab1` is followed by `Lab2`

To illustrate how these relations are used to capture information about the program under analysis, consider the following simple program:

```
   public String name (String type){
1     String a = "Anonymous";
2     if(type.equals("cat"))
3        a = "Garfield";
4     else if(type.equals("dog"))
5        a = "Snoopy";
6     else
7        a = "Blob";
8     return a;
   }
```

The input facts derived for this program are as follows:

```
assign(1, "a")   follows(1,2)   follows(4,7)
assign(3, "a")   follows(2,3)   follows(3,8)
assign(5, "a")   follows(2,4)   follows(5,8)
assign(7, "a")   follows(4,5)   follows(7,8)
```

**Task 1: Specify reaching definitions in Datalog** Specify the reaching definitions program analysis in Datalog (using the LogicBlox syntax). The resulting Datalog program must define the predicate `reach(Lab1, Var, Lab2)`, which is derived if and only if the assignment to the variable `Var` at label `Lab1` reaches label `Lab2`.

**Solution:**

```
addblock 'assign(Lab, Var) -> int(Lab), string(Var).
          follows(Lab1, Lab2) -> int(Lab1), int(Lab2).'

addblock 'reach(Lab1, Var, Lab2) <- assign(Lab1, Var), follows(Lab1, Lab2).
          reach(Lab1, Var, Lab3) <- reach(Lab1, Var, Lab2), follows(Lab2,
              Lab3), !assign(Lab2, Var).'
```

**Task 2: Find the set of reaching definitions** Show all reaching definitions for our example. To get the reaching definitions, you can run your Datalog program that you have encoded in Task 1.

**Solution:** We use the following input:

```
exec '+ assign(1, "a").
      + assign(3, "a").
      + assign(5, "a").
      + assign(7, "a").
      + follows(1,2).
      + follows(2,3).
      + follows(2,4).
```

```
      + follows(4,5).
      + follows(4,7).
      + follows(3,8).
      + follows(5,8).
      + follows(7,8).'
```

The fixed point contains the input given above, together with these tuples:

```
reach(1,"a",2)
reach(1,"a",4)
reach(3,"a",8)
reach(5,"a",8)
reach(7,"a",8)
```