A Brief Guide to Getting Through the Mysteries of Souffle (HW4)

Understanding the Data Model

We'll go through each of the commands that make our knowledge base. See `.decl` for the rules we define for you.

- `.symbol_type` This defines custom data types. This is in addition to the standard types that are in Datalog. **For this HW you only need `PersonType`, `DescriptionType`, and `number`**
- `.decl` For the section in "data model" we define for you the knowlege base that you need to accomplish these Datalog queries. Make sure you understand what these base relations are (though they are slef-explanatory):
  - `person` defines all people and their respective description
  - `female` defines all female people
  - `male` defines all male people
  - `parent_child` defines all relationships for a parent, p1, and a child, p2, of that parent
  - `person_living` defines all living people
- `.input` Accompanying each declaration we populate our knowledge base with an input command that pulls raw data from a file.

Understanding the Example Query

We'll go through each line and see what is happening.

- `.decl p0(x:PersonType, d:DescriptionType)` **For each rule that we define ourselves, we need to declare it.** Notice that we need to define 3 things: 1) its name (`p0`), 2) its internal variables (`x` and `d`), and 3) the internal variable types (`PersonType` and `DescriptionType`).
- `.output p0(IO=stdout)` This routes the output of any rule we choose to our standard output (terminal). If you choose, you may also route the output to a file by doing something like `.output p0(filename="hw2-q3-0.ans")`. Be careful of doing this if you want to save multiple versions of your answers to compare.
- `p0(x,d) :- parent_child("Priscilla",x), person(x,d).` Finally we have the acutal definition of our rule. All rule definitions should have a `:-` separator (including aggregates, more on this later). Here you can string together other rules in your defintion separated by commas. Remember that this denotes a logical AND operation. Use the same variable names in each individual atom in the definition to generate constraints such as what we see with the variable `x`. **End your definition with `.` to complete it.**

Understanding Wildcards

Wildcards (_) are used in the place of variables. Wildcards are not linked meaning that using a wildcard multiple times in the same definition does not correlate them.

Like all code we write, it should not only return the correct answer, but also be readable by other programmers. Wildcards are syntactic sugar for variables that we don't care about. This very easily conveys that for a rule where we put a wildcard instead of a varaible, we don't care about that value.

Understanding Recursion in Datalog

The essence of recursion in Datalog is to have multiple definitions of the same rule. At least one rule should tie back to the knowledge base (this "initializes" the output). The other rules should then include itself in their definition (the recursive part). How you are able to handle the recursive nature of Datalog is the main challenge of this HW. To help you start thinking of writing Datalog recursively, an example is provided:

Find all ancestors of Priscilla.

```
.decl e0(x:PersonType)
e0(x) :- parent_child(x,"Priscilla").
e0(x) :- e0(y), parent_child(x,y).
```

Understanding Query Safety (Negation)

In this HW we are utilizing stratified Datalog. This means we have negation at our disposal. To do this we can simply put `!` in front of a rule in our definition. However you should remember query safety so that for all variables in each definition, there exists a positive rule where it is defined. For example:

Find all people who are not children of Priscilla.

```
// THIS IS UNSAFE
.decl e1(x:PersonType)
e1(x) :- !parent_child("Priscilla",x).

// THIS IS SAFE (x is now defined in a positive relation "person")
.decl e2(x:PersonType)
e2(x) :- !parent_child("Priscilla",x), person(x,_).
```

Although it is possible to compile a negated wildcard in Souffle, we recommend against doing this. The semantics behind this are the same for a negated variable in that it may be "unbounded" implying that in the scope of the universe, this results of this rule blows up to infinity. For any Datalog questions on exams, we will be enforcing this idea.

Understanding Aggregations

Aggregations are a critical tool for getting through this HW. **The only aggregations you need to use for this HW are `min`, `max`, and `sum`.** The easist way to understand them is through examples. Try running these queries and/or modifying them.

Using `sum` to count. Find the count of all children of Priscilla.

```
.decl e1(n:number)
e4(n) :- n = sum(1) : parent_child("Priscilla",_).
```

Using `min` and `max`. Also how to use aggreate values in later definitions. Find the max number of children amoung Priscilla, Jezebel, and Sapphira.

```
.decl child(n:number)
.decl most_children(n:number)
child(n) :- n = sum(1) : parent_child("Priscilla",_).
child(n) :- n = sum(1) : parent_child("Jezebel",_).
child(n) :- n = sum(1) : parent_child("Sapphira",_).
most_children(m) :- m = max(c) : child(c).
```

Aggregating over multiple rules. Find the number of all living children.

```
.decl all_children(m:number)
all_children(m) :- m = sum(1) : { parent_child(_,x), person_living(x) }.
```

Grouping when aggregating to generate aggregate values correlated to a definition. Find the number of children for each parent.

```
.decl children_of_parent_all(x:PersonType, m:number)
children_of_parent_all(x,m) :- person(x,_), m = sum(1) : parent_child(x,_).
```

Code base from Dan Suciu

Written by Jonathan Leang