CS6795 Labs and Assignments Documentation

Release 1.0

Reuben Peter-Paul

October 31, 2011

CONTENTS

CHAPTER

ONE

ASSIGNMENT 0

Todo

Using only a pencil and the left half of a paper, construct a random, non-trivial well-formed XML element with tag names x, y, z and sub(...sub)elements as follows: Write an x/y/z start-tag such as <x>, pronouncing it “angle, x, angle”; leave plenty of space and write, vertically below, the matching end-tag </x>, pronouncing it “angle, slash, x, angle” (with practice, you can also pronounce the ‘x-colored brackets’ <x> as “start-x” and </x> as “end-x”). Then fill in, indented by two blanks, another x/y/z start-tag such as <y>, pronouncing it “angle, y, angle” (“start-y”); leave some space and write, again vertically below, the matching end-tag </y>, pronouncing it “angle, slash, y, angle” (“end-y”). If there is more space left below the current subelement, proceed with the next subelement vertically below it; otherwise, proceed by filling in the space between two other pairs of matching tags, indented by two further blanks. Continue in this way to fill in the space between matching pairs of tags, repeatingly using tag names from the set {x, y, z}. However, instead of adding more tag pairs, you may also fill in natural-language phrases between matching pairs of tags.

1.1 Parts A, B

Todo

- Count the number of subelements of your generated XML element, including the main element in the sum. For each tag name x, y, z give the number of subelements using it.
- Write a Prolog term on the right half of your paper such that an XML tag pair like <x> ... </x> becomes a Prolog structure x(...). Align each constructor and its opening parenthesis such as x(with the corresponding start-tag <x>; align each matching closing parenthesis) with the corresponding end-tag </x>. Put an XML natural-language phrase into double quotes (”) before its use in a Prolog structure.

1.2 Part C

Todo

Draw the node-labeled, (left-to-right-)ordered tree for which the XML element and equivalent Prolog structure are just two linearized representations. Hint: The vertical subelement/substructure extension corresponds to subtree breadth; the horizontal subelement/substructure indention corresponds to subtree depth.

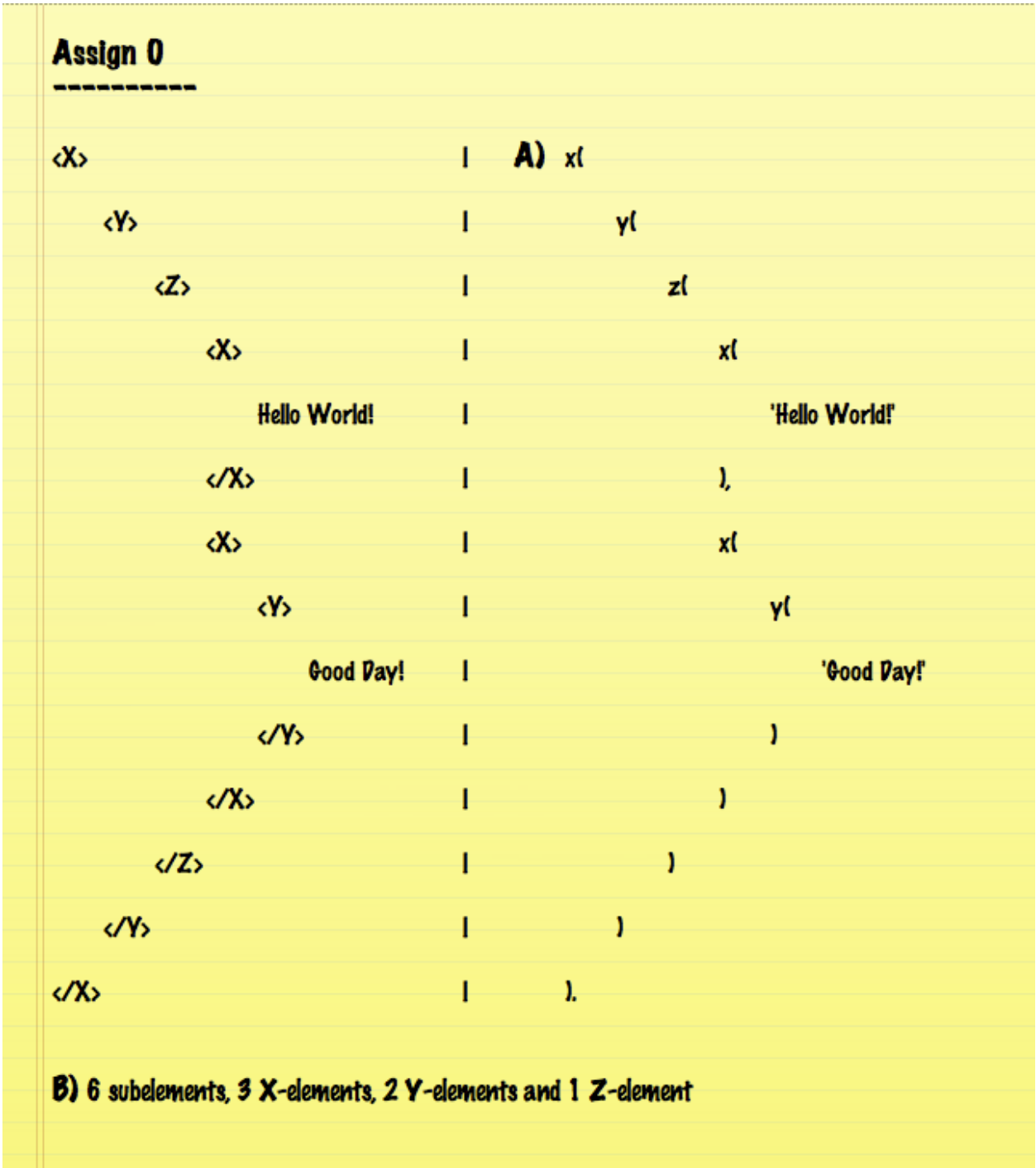
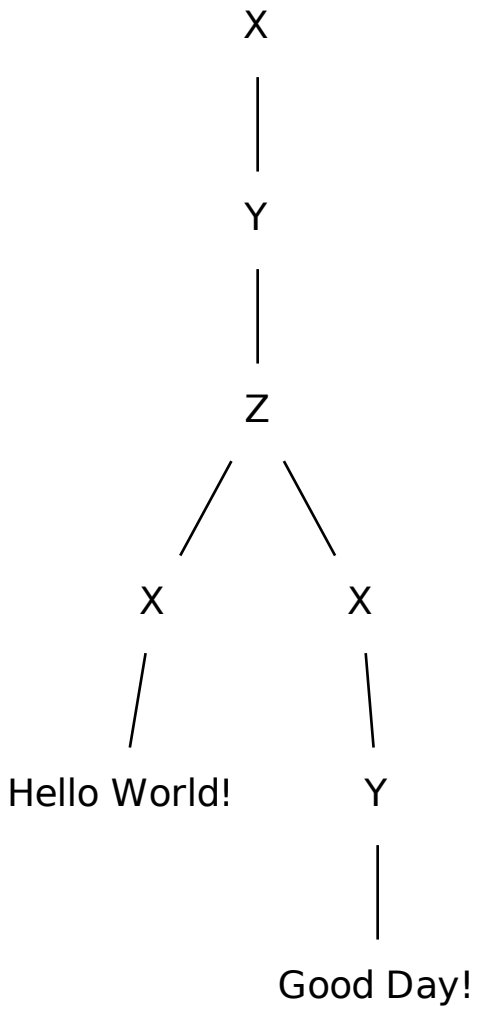


Figure 1.1: *Left:* Construction of a non-trivial *well-formed* XML instance using tag names `x`, `y`, and `z`. *Right:* Prolog term *pretty-printed* to align with the XML tag elements on the left. *Bottom:* Answer to part **A** of assignment 0.



Visualization of the XML isntance tree structure, *rendered using Graphviz*.

1.3 Part D

Todo

List the notational (dis)advantages of the XML and Prolog representations.

The notational advantages of xml representations are generally well known among programmers and are many (cf. [JSON]):

- XML is human readable.

- XML can be used as an exchange/interchange format to enable users to mover thier data between similar applications.
- XML provides structure to data so taht it is richer in information.
- XML is easily processed because of the structure of the data is simple and standard.
- There is a wide range of reusable software available to programmers to handle XML so they don't have to re-invent code.
- XML seperates the presentation of data from the structure of that data.
- Many views of the one data.
- Self-describing data.
- Complete integration of all traditional databases and formats.
- Internationalization.
- Open and extensible.
- XML is widely adopted by the computer industry.

The notational disadvantages of xml representations are also generally well know to programmers (cf. [Tolf11]):

- DOM is too specialized.
- Can be cumbersome and inefficient.
- Does not map well to data types of most programming languages.

The notational advantages of prolog representations are similar to those of XML in that they are human readable, they provide a structure to the data so that it is richer in information.

The notational disadvantages of prolog are inherent to all dynamically checked programming languages, e.g.: type checking done at run-time, only one datatype (`term`) (cf. [Wp1]).

1.4 Part E

Todo

Can anything be logically wrong with the legal Prolog structures corresponding to arbitrary XML elements that use repeated tags from {x, y, z}? Hint: Consider ways in which not only XML but also Prolog is "less formal" than logic (types, modes, arities, ...).

XML and Prolog are less formal and their type specifications are much more relaxed than formal logics. According to [Brac04] (chapter 2, p. 25) the aim of formal logics is to build up a *Logical Model* such that the set of possible interpretations is made more narrow so as to rule out more and more unintended interpretations. Ultimately, logical consequene itself will tend toward "'truth in the intended interpretation"'. The *document object mode (DOM)* implemented by XML does not provide semantics for "logical implications/inference/entailment" the relationships between elements are structural only, and while a term can be expressed in *Prolog* that mimics the structure of an XML document queries against such terms are seem to be *meaningless* and *useless* since the inner terms are inaccessbile to the `top loop`.

Prolog REPL Sample

```
?- ['Assign0.pl'].
% Assign0.pl compiled 0.00 sec, 1,768 bytes
true.

?- listing.

a(b(c('hello world'))).

x(y(z(x('hello world'), x(y(hi))))).
true.

?- c(X).
ERROR: toplevel: Undefined procedure: c/1 (DWIM could not correct goal)
?- z(X).
ERROR: toplevel: Undefined procedure: z/1 (DWIM could not correct goal)
```

CHAPTER

TWO

LAB 1 - CREATE YOUR OWN XML DTD, XSD, AND RNC OR RNG

Todo

Consider these examples of XML documents for clause sets consisting of zero or more facts f (Prolog: `f.`) and zero or more 'backward' rules c <- p (Prolog: `c :- p.`), in any order ("myurl" will be replaced as explained below):

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE clauses SYSTEM "myurl">
 <clauses>
    <fact> f </fact>
    <rule> <conc> c </conc> <prem> p </prem> </rule>
 </clauses>

<?xml version="1.0" standalone="no"?>
<!DOCTYPE clauses SYSTEM "myurl">
<clauses>
    <rule> <conc> c1 </conc> <prem> p1 </prem> </rule>
    <fact> f1 </fact>
    <rule> <conc> c2 </conc> <prem> p2 </prem> </rule>
    <fact> f2 </fact>
    <fact> f3 </fact>
</clauses>
```

Inductively complete this XML DTD (overwrite the "..." lines) for such clause sets:

```
<!ELEMENT clauses   (.................)>
<!ELEMENT rule   (.................)>
<!ELEMENT fact    (#PCDATA)>
<!ELEMENT ........    (.................)>
<!ELEMENT ........    (.................)>
```

2.1 DTD

```
<!ELEMENT clauses (fact|rule)*>
<!ELEMENT rule                (conc,prem)>
<!ELEMENT fact                (#PCDATA)>
<!ELEMENT conc                (#PCDATA)>
<!ELEMENT prem                (#PCDATA)>
```

2.2 XSD

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="clauses">
    <xs:complexType>
      <xs:sequence>
        <xs:choice maxOccurs="unbounded">
          <xs:element ref="rule"/>
          <xs:element ref="fact"/>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="rule">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="conc"/>
        <xs:element ref="prem"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="fact" type="xs:string"/>
  <xs:element name="conc" type="xs:string"/>
  <xs:element name="prem" type="xs:string"/>
</xs:schema>
```

2.3 RelaxNG

Compact syntax:

```
default namespace = ""

start = clauses
clauses = element clauses {
  (element rule {
    (element conc {xsd:string},
    element prem {xsd:string})}
  | (element fact {xsd:string}))*}
```

Note: I used trang to transform RelaxNG compact syntax into xml-based syntax.

XML syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar ns="" xmlns="http://relaxng.org/ns/structure/1.0" datatypeLibrary="http://www.w3.org/2001/XM
  <start>
    <ref name="clauses"/>
  </start>
  <define name="clauses">
    <element name="clauses">
      <zeroOrMore>
        <choice>
          <element name="rule">
```

```
            <element name="conc">
              <data type="string"/>
            </element>
            <element name="prem">
              <data type="string"/>
            </element>
          </element>
          <element name="fact">
            <data type="string"/>
          </element>
        </choice>
      </zeroOrMore>
    </element>
  </define>
</grammar>
```

2.4 Misc

Note: To validate the above sample instances I used xmllint - command line XML tool to parse and type-check/validate:

```
$ xmllint --noout --schema http://reubenpeterpaul.github.com/lab1/XSD/clauses.xsd clauses-instance.xm
$ xmllint --noout --dtdvalid http://reubenpeterpaul.github.com/lab1/DTD/clauses.dtd clauses-instance.
$ trang clauses.rnc clauses.rng
$ xmllint --noout --relaxng http://reubenpeterpaul.github.com/lab1/DTD/clauses.rng clauses-instance.x
```

Note: I also used my own custom XML catalog file:

```
<?xml version="1.0"?>
<!DOCTYPE catalog
PUBLIC "-//OASIS/DTD Entity Resolution XML Catalog V1.0//EN"
"http://www.oasis-open.org/committees/entity/release/1.0/catalog.dtd">
<catalog
xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">

...

  <system
  systemId="http://reubenpeterpaul.github.com/cs6795/lab1/DTD/clauses.dtd"
  uri="/home/peter-paulr/.laboratory/cs6795/lab1/clauses.dtd"
  />

  <system
  systemId="http://reubenpeterpaul.github.com/cs6795/lab1/XSD/clauses.xsd"
  uri="/home/peter-paulr/.laboratory/cs6795/lab1/clauses.xsd"
  />

  <system
  systemId="http://reubenpeterpaul.github.com/cs6795/lab1/RNG/clauses.rng"
  uri="/home/peter-paulr/.laboratory/cs6795/lab1/clauses.rng"
  />
</catalog>
```

CHAPTER

THREE

LAB 2 - GENERATE RDF GRAPH

Todo

Go to the W3C RDF Validation Service Copy and edit an RDF serialization of your choice over the example in the text field or just use the example itself, at "Display Result Options:" select "Triples and Graph", and Hit the 'Parse RDF:' button.

Note: I used the following RDF/XML document:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:a="http://www.w3.org/2001/svgRdf/axsvg-schema.rdf#">
  <rdf:Description rdf:about="#MyList">
    <a:consistsOf rdf:parseType="Collection">
      <rdf:Description rdf:about="#a"/>
      <rdf:Description rdf:about="#b"/>
      <rdf:Description rdf:about="#c"/>
      <rdf:Description rdf:about="#d"/>
    </a:consistsOf>
  </rdf:Description>
</rdf:RDF>
```

The following graph was generated:

Figure 3.1: The result of parsing the above RDF/XML document and generating a graph depicting the a `Lisp` style `list`.

CHAPTER

FOUR

ASSIGNMENT 1

Note: Please mark after parts *Part A-1* and *Part A-2*.

4.1 Part A-0

Todo

Do the missing leaf data for passengers cause any (XML-level) issue?

No the the missing leaf data (`#PCDATA`) does not affect the xml well-formedness.

4.2 Part A-1

Todo

Does the repeatedly occurring `appellation`:

```
appellation('Main St').     appellation('Fredericton').
```

label cause a problem with respect to the unambiguous representation of all parts of the taxi-ride information? Hint: Is it even necessary to check whether every leaf of the tree - ordered from 37 to 12.50 - are uniquely denoted by the "path name" of node labels leading to it from the taxi-ride root? What about any other repeatedly occurring label?

To determine potential ambiguity introduced by the repeated use of the `appellation` tag-label consider the following prolog system:

```
appellation('Main St').
appellation('Fredericton').
first('Peter').
mid('C.').
lastn('Jones').
numbern(37).
numbern(12).
province('NB').

name(X,Y,Z) :- first(X), mid(Y), lastn(Z).
driver(W,X,Y,Z) :- numbern(W), name(X,Y,Z).
```

```
municipality(X,Y) :- appellation(X), province(Y).
street(X,Y) :- number(X), appellation(Y).
```

Backtracking in prolog against the query `street(X,Y)` yields the following results:

```
?- ['testA2'].
% testA2 compiled 0.00 sec, 5,624 bytes
true.

?- street(X,Y).
X = 37,
Y = 'Main St' ;
X = 37,
Y = 'Fredericton' ;
X = 12,
Y = 'Main St' ;
X = 12,
Y = 'Fredericton'.
```

This illustrates by example that the repeated use of `appellation` and also `number` results in adding ambiguity to the representation of `taxi-ride`. This problem can be dealt with via introduction of complex types, or denoting the leaf nodes with the path of node-labels from the root `taxi-ride`, e.g.:

```
street(numbern(12), appellation('Main St')).
municipality(appellation('Fredericton'), province('NB')).
```

4.3 Part A-2

Todo

What if the nodes labeled 'appellation' would have been labeled 'name', too?

Again let us consider a prolog system:

```
name('Main St').
name('Fredericton').
first('Peter').
mid('C.').
lastn('Jones').
numbern(37).
numbern(12).
province('NB').

name(X,Y,Z) :- first(X), mid(Y), lastn(Z).
driver(W,X,Y,Z) :- numbern(W), name(X,Y,Z).
municipality(X,Y) :- name(X), province(Y).
street(X,Y) :- number(X), name(Y).
```

Backtracaking in prolog REPL against the query `street` and `driver` yields the following results:

```
?- consult('testA2b').
% testA2b compiled 0.00 sec, 5,096 bytes
true.


?- street(X,Y).
```

```
X = 37,
Y = 'Main St' ;
X = 37,
Y = 'Fredericton' ;
X = 12,
Y = 'Main St' ;
X = 12,
Y = 'Fredericton'.

?- driver(W,X,Y,Z).
W = 37,
X = 'Peter',
Y = 'C.',
Z = 'Jones'.
W = 12,
X = 'Peter',
Y = 'C.',
Z = 'Jones'.
```

The result is the same as above (*Part A-1*). `Node-labelled` paths are still required to denotate the `name` tags used by `street` and `municipality` nodes. While the rule for `name` satisfies the complex conditions for separating the parts of an individuals name.

4.4 Part A-3 - Relabeling of tree in Prolog

```
carnumber(37).
firstname('Peter').
middlename('C.').
lastname('Jones').
date('9/27').
streetnumber(12).
streetname('Main St').
municipalityname('Fredericton').
provincename('NB').
fare(12.50).

taxiride(W,X,Y,Z,D,E,F,G,H,I) :-
  driver(W,X,Y,Z),
  passengers,
  transportinformation(D,E,F,G,H,I).

driver(W,X,Y,Z) :- carnumber(W), fullname(X,Y,Z).

fullname(X,Y,Z) :- firstname(X), middlename(Y), lastname(Z).

transportinformation(D,E,F,G,H,I) :- date(D), destination(E,F,G,H),
fare(I).

destination(E,F,G,H) :- street(E,F), municipality(G,H).

street(X,Y) :- streetnumber(X), streetname(Y).

municipality(X,Y) :- municipalityname(X), provincename(Y).
```

4.5 Part B-1 - Well formed XML element from *Part A-1*

```
<?xml version="1.0" ?>
<taxi-ride>
  <driver>
    <number>37</number>
    <name>
      <first>Peter</first>
      <mid>C.</mid>
      <last>Jones</last>
    </name>
  </driver>
  <passengers />
  <tranport-information>
    <date>9/27</date>
    <street>
      <number>12</number>
      <appellation>Main St</appellation>
    </street>
    <municipality>
      <appellation>Fredericton</appellation>
      <province>NB</province>
    </municipality>
    <fare>
      12.50
    </fare>
  </tranport-information>
</taxi-ride>
```

Yes, a well formed XML instance can be given for *Part A-1*

4.6 Part B-2 - Well formed XML element from *Part A-2*

```
<?xml version="1.0" ?>
<taxi-ride>
  <driver>
    <number>37</number>
    <name>
      <first>Peter</first>
      <mid>C.</mid>
      <last>Jones</last>
    </name>
  </driver>
  <passengers />
  <tranport-information>
    <date>9/27</date>
    <street>
      <number>12</number>
      <name>Main St</name>
    </street>
    <municipality>
      <name>Fredericton</name>
      <province>NB</province>
    </municipality>
    <fare>
      12.50
```

```
    </fare>
  </tranport-information>
</taxi-ride>
```

Yes, a well formed XML instance can be given for *Part A-2*

4.7 Part B-3 - XML instance of relabeling

```
<taxiride>
  <driver>
    <carnumber>37</carnumber>
    <fullname>
      <firstname>Peter</firstname>
      <middlename>C.</middlename>
      <lastname>Jones</lastname>
    </fullname>
  </driver>
  <passengers />
  <transportinformation>
    <date>9/27</date>
    <destination>
      <street>
        <streetnumber>12</streetnumber>
        <streetname>Main St</streetname>
      </street>
      <municipality>
        <municipalityname>Fredericton</municipalityname>
        <provincename>NB</provincename>
      </municipality>
    </destination>
    <fare>12.50</fare>
  </transportinformation>
</taxiride>
```

4.8 Part C - Prolog equivalent term of *Part B-3 - XML instance of relabeling*

```
taxiride(
  driver(
    carnumber(37),
    fullname(
      firstname('Peter'),
      middlename('C.'),
      lastname('Jones'))),
  passengers,
  transportinformation(
    date('9/27'),
    destination(
      street(
        streetnumber(12),
        streetname('Main St')),
      municipality(
        municipalityname('Fredericton'),
```

```
        provincename('NB'))),
    fare(12.50))).
```

4.9 Part D-1 - DTD schema for *Part A-1*

```
<!ELEMENT taxi-ride             (driver,passengers,transport-information)>
<!ELEMENT driver                (number,name)>
<!ELEMENT name                  (first,mid,last)>
<!ELEMENT number                (#PCDATA)>
<!ELEMENT first                 (#PCDATA)>
<!ELEMENT mid                   (#PCDATA)>
<!ELEMENT last                  (#PCDATA)>
<!ELEMENT passengers            (#PCDATA)>
<!ELEMENT transport-information (date,destination,fare)>
<!ELEMENT date                  (#PCDATA)>
<!ELEMENT destination           (street,municipality)>
<!ELEMENT street                (number,appellation)>
<!ELEMENT municipality          (appellation,province)>
<!ELEMENT appellation           (#PCDATA)>
<!ELEMENT province              (#PCDATA)>
```

It is possible to write a DTD that precisely describe *Part A-1* and *Part B-1 - Well formed XML element from a1* since when DTD's are designed/applied semantics are not considered only the structure of the DOM-tree is being restricted.

4.10 Part D-2 - DTD schema for *Part A-2*

```
<!ELEMENT taxi-ride             (driver,passengers,transport-information)>
<!ELEMENT driver                (number,name)>
<!ELEMENT name                  ((first,mid,last)|#PCDATA)>
<!ELEMENT number                (#PCDATA)>
<!ELEMENT first                 (#PCDATA)>
<!ELEMENT mid                   (#PCDATA)>
<!ELEMENT last                  (#PCDATA)>
<!ELEMENT passengers            (#PCDATA)>
<!ELEMENT transport-information (date,destination,fare)>
<!ELEMENT date                  (#PCDATA)>
<!ELEMENT destination           (street,municipality)>
<!ELEMENT street                (number,name)>
<!ELEMENT municipality          (name,province)>
<!ELEMENT province              (#PCDATA)>
<!ELEMENT fare                  (#PCDATA)>
```

It is not possible to write a DTD that describes *Part A-1* or *Part B-1 - Well formed XML element from a1* since `((first,mid,last)|#PCDATA)` introduces ambiguity into the structure and is not permitted as a valid DTD syntax.

4.11 Part D-3 - DTD schema for *a3*

```
<!ELEMENT taxiride              (driver,passengers,transportinformation)>
<!ELEMENT driver                (carnumber,fullname)>
<!ELEMENT carnumber             (#PCDATA)>
```

```
<!ELEMENT fullname              (firstname,middlename,lastname)>
<!ELEMENT firstname             (#PCDATA)>
<!ELEMENT middlename            (#PCDATA)>
<!ELEMENT lastname              (#PCDATA)>
<!ELEMENT passengers            (#PCDATA)>
<!ELEMENT transportinformation  (date,destination,fare)>
<!ELEMENT date                  (#PCDATA)>
<!ELEMENT destination           (street,municipality)>
<!ELEMENT street                (streetnumber,streetname)>
<!ELEMENT streetnumber          (#PCDATA)>
<!ELEMENT streetname            (#PCDATA)>
<!ELEMENT municipality          (municipalityname,provincename)>
<!ELEMENT municipalityname      (#PCDATA)>
<!ELEMENT provincename          (#PCDATA)>
<!ELEMENT fare                  (#PCDATA)>
```

The *Part B-3 - XML instance of relabeling*-element is *Structurally* valid w.r.t. to the above DTD.

CHAPTER

FIVE

ASSIGNMENT 2

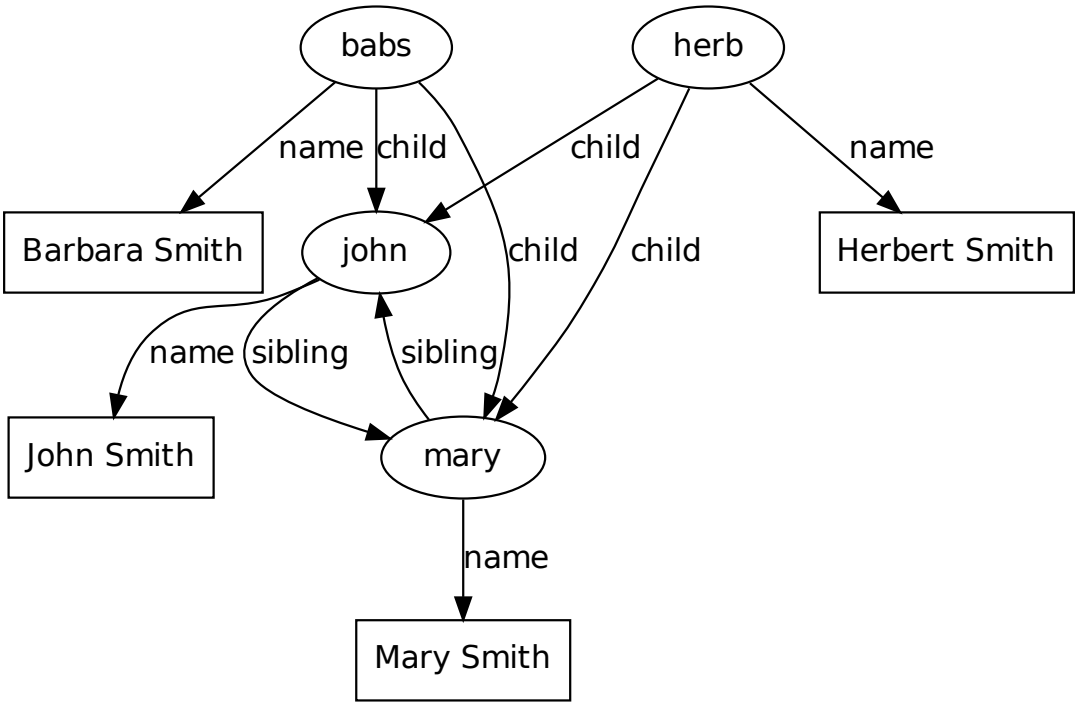
5.1 Part 1 - Draw DLG of RDF/XML Document

Todo

This is RDF metadata about four fictitious people:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:samfam="http://samplefamily.org/elements/">
  <rdf:Description rdf:about="http://www.ourhomes/john">
    <samfam:name>John Smith</samfam:name>
    <samfam:sibling rdf:resource="http://www.ourhomes/mary"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.ourhomes/mary">
    <samfam:name>Mary Smith</samfam:name>
    <samfam:sibling rdf:resource="http://www.ourhomes/john"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.ourhomes/babs">
    <samfam:name>Barbara Smith</samfam:name>
    <samfam:child rdf:resource="http://www.ourhomes/john"/>
    <samfam:child rdf:resource="http://www.ourhomes/mary"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.ourhomes/herb">
    <samfam:name>Herbert Smith</samfam:name>
    <samfam:child rdf:resource="http://www.ourhomes/john"/>
    <samfam:child rdf:resource="http://www.ourhomes/mary"/>
  </rdf:Description>
</rdf:RDF>
```

Draw the directed labeled graph (DLG) that constitutes the RDF diagram of this XML element (use space below). Hint: URLs, going into ovals or becoming arc labels, and texts, going into rectangles, may be arbitrarily shortened, along as they remain unique (e.g.: '.../john' or just 'john'; 'John S' or just 'JS'); namespaces can be omitted.



Note: Above graph created manually using the following graphviz DOT language:

```
digraph g {

  // Resources
  node [label="john"] john;
  node [label="mary"] mary;
  node [label="babs"] babs;
  node [label="herb"] herb;


  // Literals
  node [shape=rectangle, label="John Smith"] john_name;
  node [shape=rectangle, label="Mary Smith"] mary_name;
  node [shape=rectangle, label="Barbara Smith"] babs_name;
  node [shape=rectangle, label="Herbert Smith"] herb_name;

  // Name relationships
  john -> john_name [label="name"];
  mary -> mary_name [label="name"];
  babs -> babs_name [label="name"];
  herb -> herb_name [label="name"];

  // sibling relationships
  john -> mary [label="sibling"];
  mary -> john [label="sibling"];
```

```
  // Child relationships
  babs -> mary [label="child"];
  babs -> john [label="child"];
  herb -> mary [label="child"];
  herb -> john [label="child"];

}
```

5.2 Part 2

Todo

Consider the following Datalog program in Prolog syntax defining a unary predicate or relation human:

```
human(X) :- philosopher(X).
human(X) :- featherless(X), biped(X).
philosopher(X) :- dualist(X).
dualist(john).
biped(mary).
```

Todo

1. Give, and very briefly explain, the result of the query human(john):

yes **because** *''john'' is a ''dualist'' and dualists are ''philosophers'' and philosophers are ''human'', therefore ''john'' is a ''human''*

Todo

2. Give, and very briefly explain, the result of the query human(mary):

no **because** *''mary'' is a ''biped'', but only human if a ''biped'' and ''featherless'' however it is not known if she is a ''featherless''. Therefore, it is not known that she is human.*

Todo

c) Give the result(s) of the query: .. code-block:

```
?- human(Y).
```

Y = john no

5.3 Part 3

Todo

Using Prolog or any other logic notation, give a program that expresses that (1) `ride(X,Y)` can be proved via `train(X,Y)` or `bus(X,Y)` and (2) `ride(X,Z)` can be proved via (a) `train(X,Y)` or `bus(X,Y)` and (b), recursively, `ride(Y,Z)`. Add facts representing `train` or `bus` relations in a real or fictitious region, mentioning a

place, one of its `train`- or `bus`-reachable places, and one of its further reachable places. Show a query that asks for all of the known `rides`, and a derivation using at least one occurrence of the rule (2).

Hints: Consider to introduce an auxiliary relation for direct rides. You can test your program and queries in the SWI Prolog engine (variables are upper-cased) or in the OO jDREW POSL engine (variables are "?"-prefixed).

```
ride(X,Y) :- train(X,Y).
ride(X,Y) :- bus(X,Y).
ride(X,Z) :-
  train(X,Y),
  ride(Y,Z).
ride(X,Z) :-
  bus(X,Y),
  ride(Y,Z).

bus(home,kings_place).
bus(kings_place,work).
bus(kings_place,train_station).
train(train_station,bathurst).

?- ride(X,Y).
X = train_station,
Y = bathurst ;
X = home,
Y = kings_place ;
X = kings_place,
Y = work ;
X = kings_place,
Y = train_station ;
X = home,
Y = work ;
X = home,
Y = train_station ;
X = home,
Y = bathurst ;
X = kings_place,
Y = bathurst ;
false.

?- ride(home,bathurst).
true ;
false.

?- ride(home,work).
true ;
false.
```

5.4 Part 4

Todo

Consider this definition of the predicate `goldmemgold` (i.e., 'member surrounded by gold'):

Prolog syntax:

```
goldmemgold(X,[gold,X,gold|R]).
goldmemgold(X,[Y|R]) :- goldmemgold(X,R).
```

POSL syntax:

```
goldmemgold(?X,[gold,?X,gold|?R]).
goldmemgold(?X,[?Y|?R]) :- goldmemgold(?X,?R).
```

You can read the two clauses as follows: ** X is a ‘‘goldmemgold‘‘ of a list whose first three elements are gold followed X followed by gold. * X is a ‘‘goldmemgold‘‘ of a list whose tail (all but the first element) is R if X is a goldmemgold of R.*

Hint: You can test your answers to the following in the SWI Prolog engine (variables are upper-cased) or in the OO jDREW POSL engine (variables are "?"-prefixed).

Show the results of checking for a specific goldmemgold thus:

```
?- goldmemgold(john,[john,gold,mary,gold,peter,gold]).
false.

?- goldmemgold(mary,[john,gold,mary,gold,peter,gold]).
true ;
false.
```

Todo

Show the results of enumerating goldmemgolds of a list thus (where ";" asks for another solution):

Prolog syntax:

```
?- goldmemgold(X,[john,gold,mary,gold,peter,gold]).
```

POSL syntax:

```
?- goldmemgold(?X,[john,gold,mary,gold,peter,gold]).
```

```
?- goldmemgold(X,[john,gold,mary,gold,peter,gold]).
X = mary ;
X = peter ;
false.
```

Todo

Briefly explain here the number of solutions found, e.g. by studying the expanded proof tree under "Solution:" in the GUI of OO jDREW.

The number of solutions found by studying the expanded proof tree under "solution:" in the GUI of OO jDREW can be explained by the nature of how `prolog` iteratest through the list:

```
[john,gold,mary,gold,peter,gold]
```

The proof tree for `mary` contains **one** recursive step effectively popping `john` off the front of the list.

The proof tree for `peter` contains **three** recursive steps effectively popping `john`, `gold`, and `mary` off of the front of the list successively until `goldmemgold(peter,[gold,peter,gold|?R])` succeeds.

Todo

Briefly explain if it has any effect on the number of solutions that john, mary, and peter 'share' some gold?

Yes **because** if `john`, `mary` and `peter` did not 'share' some `gold` the proof for `mary` would require **one** additional recursive step to reduce the list `[john,gold,gold,mary,gold,...]` down to `[gold,mary,gold,...]` and an additional **two** recursive steps to 'reduce' `[john,gold,gold,mary,gold,gold,peter,gold]` down to `[gold,peter,gold]`.

CHAPTER

SIX

ASSIGNMENT 3

6.1 Part 1

Todo

1. This is a DTD for simple (almost natural-language) 'forward' rules and facts:

```
<!ELEMENT forward    ((rule | fact)*)>
<!ELEMENT rule          (if, then)>
<!ELEMENT fact          (#PCDATA)>
<!ELEMENT if             (#PCDATA)>
<!ELEMENT then         (#PCDATA)>
```

a) Are the following XML elements valid with respect to this DTD (write "yes" or "no" behind them)?

```
<forward> </forward>
<!-- YES -->

<forward> the weather </forward>
<!-- NO -->

<forward>
  <fact> it snows </fact>
</forward>
<!-- YES -->

<forward>
  <rule> if it rains then the road gets wet </rule>
</forward>
<!-- NO -->

<forward>
  <rule>
    <if> it rains </if>
    <then> the road gets wet </then>
  </rule>
</forward>
<!-- YES -->

<forward>
  <fact> it rains </fact>
  <rule>
```

```
    <if> it rains </if>
    <then> the road gets wet </then>
  </rule>
</forward>
<!-- YES -->
```

Todo

2. Consider these XML elements for 'forward' (p => c) and 'backward' (c <= p) rules:

```
<forward>
  <rule>
    <if> p </if>
    <then> c </then>
  </rule>
</forward>

<backward>
  <rule>
    <conc> c </conc>
    <prem> p </prem>
  </rule>
</backward>
```

Inductively complete this XML DTD (write into the "..." lines) for 'backward' rules and facts:

```
<!ELEMENT backward    ((rule|fact)*)>
<!ELEMENT rule             (conc, prem)>
<!ELEMENT fact           (#PCDATA)>
<!ELEMENT conc           (#PCDATA)>
<!ELEMENT prem           (#PCDATA)>
```

6.2 Part 2

Todo

Complete the following XSLT template – by just filling in the "..." versions – for the (XML-to-XML) transformation of 'forward' rules and facts into 'backward' rules and facts:

```
<xsl:template match="forward">
  <backward>
    <xsl:apply-templates/>
  </backward>
</xsl:template>

<xsl:template match="rule">
  <rule>
    <conc><xsl:value-of select="then"/></conc>
    <prem><xsl:value-of select="if"/></prem>
  </rule>
</xsl:template>

<xsl:template match="fact">
  <fact>
```

```
    <xsl:value-of select="."/>
  </fact>
</xsl:template>
```

6.2.1 Transformation inversion?

Todo

Could this transformation be 'inverted' mapping 'backward' rules and facts into 'forward' rules and facts without information loss (write in "yes" or "no" here)?

Yes.

6.3 Part 3

Again consider the following Datalog program in Prolog syntax:

```
human(X) :- philosopher(X).
human(X) :- featherless(X), biped(X).
philosopher(X) :- dualist(X).
dualist(john).
biped(mary).
```

Todo

1. Give its grounding (consistently replacing variables by constants in each rule):

```
human(john) :- philosopher(john).
human(mary) :- philosopher(mary).
human(john) :- featherless(john), biped(john).
human(mary) :- featherless(mary), biped(mary).
philosopher(john) :- dualist(john).
philosopher(mary) :- dualist(mary).
dualist(john).
biped(mary).
```

Todo

Note: Shortcut of the grounded program:

```
h1 :- p1.
h2 :- p2.
h1 :- f1, b1.
h2 :- f2, b2.
p1 :- d1.
p2 :- d2.
d1.
b2.
```

M = {d1,b2,p1,h1}

---

2. Construct its Least Herbrand Model by fixpoint iteration (starting with the set of facts, applying the rules bottom-up to add new facts, etc., until the set no longer changes):

Fixpoint iteration:

- Step 1:

  M0 = {d1, b2}

- Step 2:

  M1 = {d1, b2} + {p1}

- Step 3:

  M2 = {d1, b2, p1} + {h1}

Least Herbrand Model is: M = {dualist(john), biped(mary), philosopher(john), human(john)}

6.4 Part 4

Todo

Using a knowledge base with the following facts and rules about fictitious people, employ OO jDREW to query their represented social network.

Note: see Figure below.

```
knows_from_highschool(Mary, John).

knows_from_highschool(John, Peter).

knows_from_university(Peter, Cora).

knows_from_university(Cora, Gisele).


knows(?X, ?Y) :- knows_from_highschool(?X, ?Y).

knows(?X, ?Y) :- knows_from_university(?X, ?Y).


knows_trans(?X, ?Y) :- knows(?X, ?Y).

knows_trans(?X, ?Y) :-  knows(?X, ?Z),
knows_trans(?Z, ?Y).
```

Give all results of the following (top-down) queries employing OO jDREW TD:

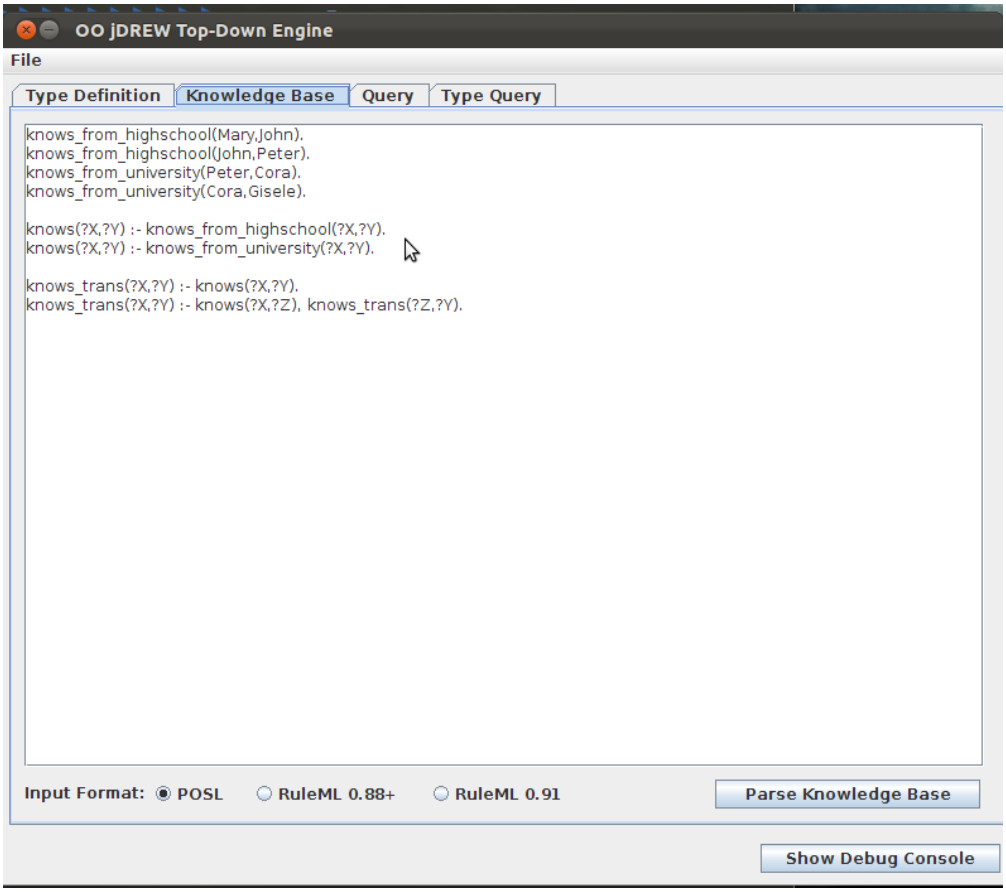
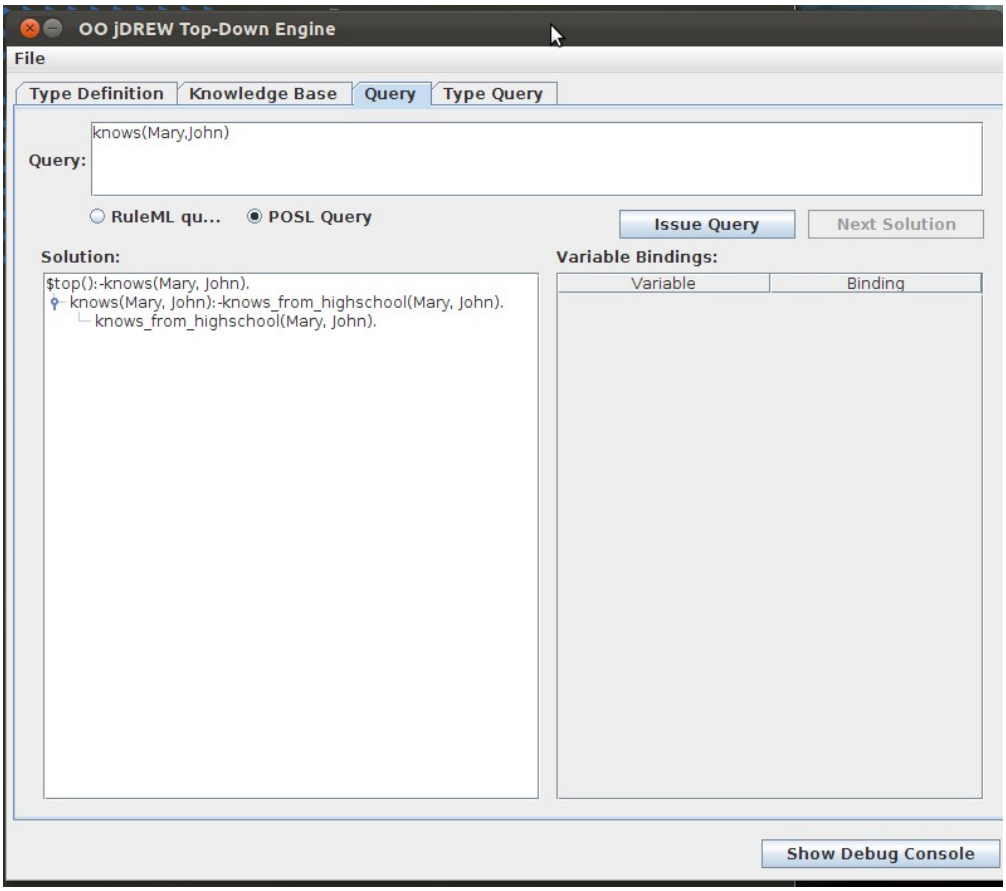
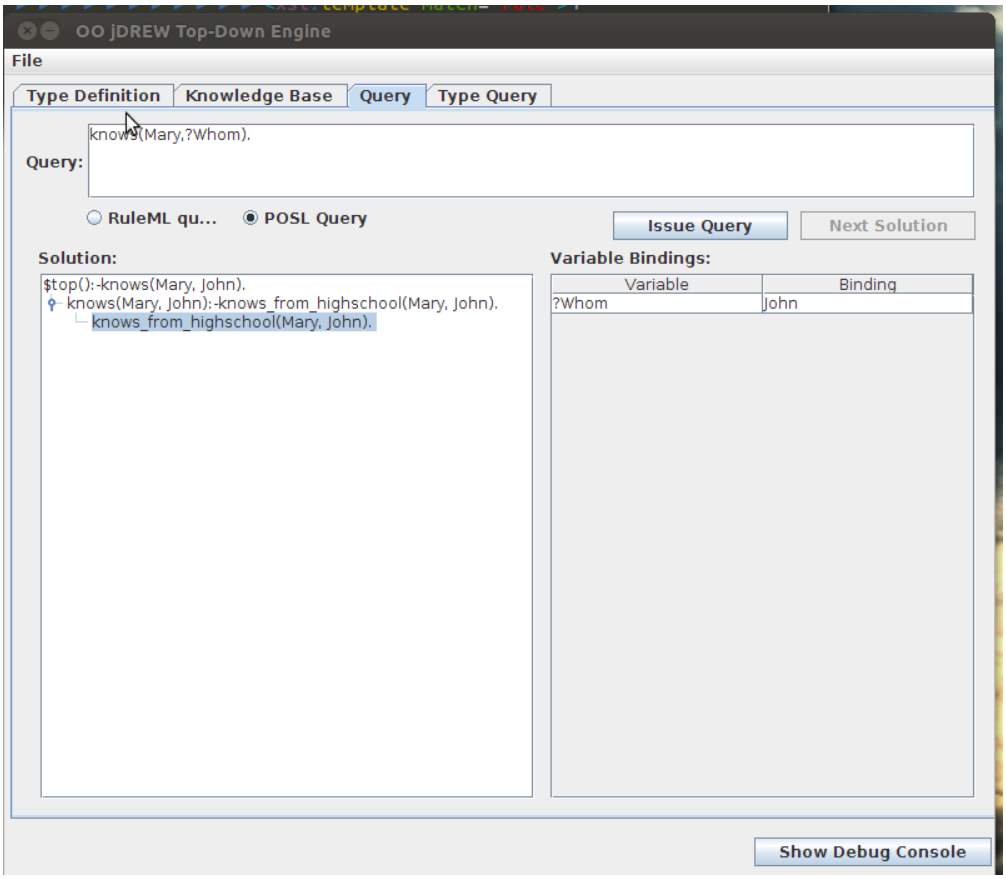


Figure 6.1: Screenshot of the knowledge base entered into OO jDREW Top-Down Engine.


OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows(Mary,John)
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows(Mary, John).
knows(Mary, John):-knows_from_highschool(Mary, John).
knows_from_highschool(Mary, John).
Variable Bindings:
Variable
Binding
Show Debug Console

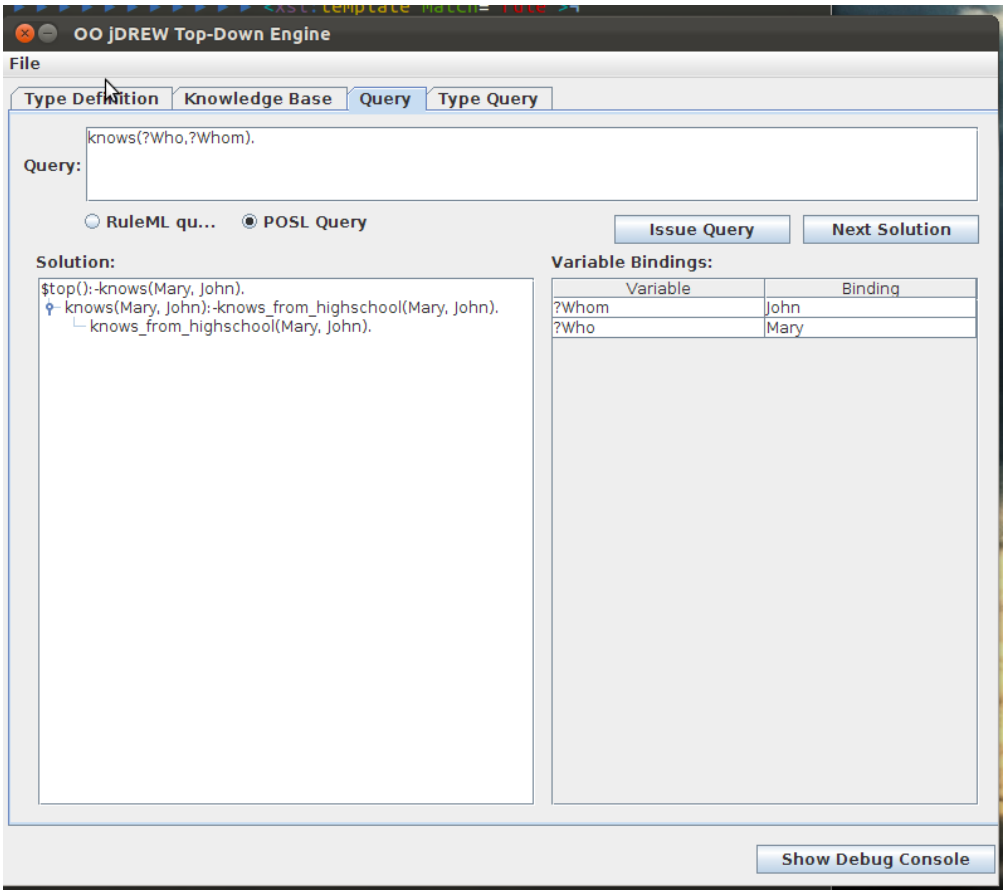

OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows(Mary,?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows(Mary, John).
knows(Mary, John):-knows_from_highschool(Mary, John).
knows_from_highschool(Mary, John).
Variable Bindings:
Variable
Binding
?Whom
John
Show Debug Console

6.4.1 knows(Mary, John)

6.4.2 knows(Mary, ?Whom)

6.4.3 knows(?Who, ?Whom)

See following 4 screen shots.



–

–

–

6.4.4 knows_trans(Mary, John)

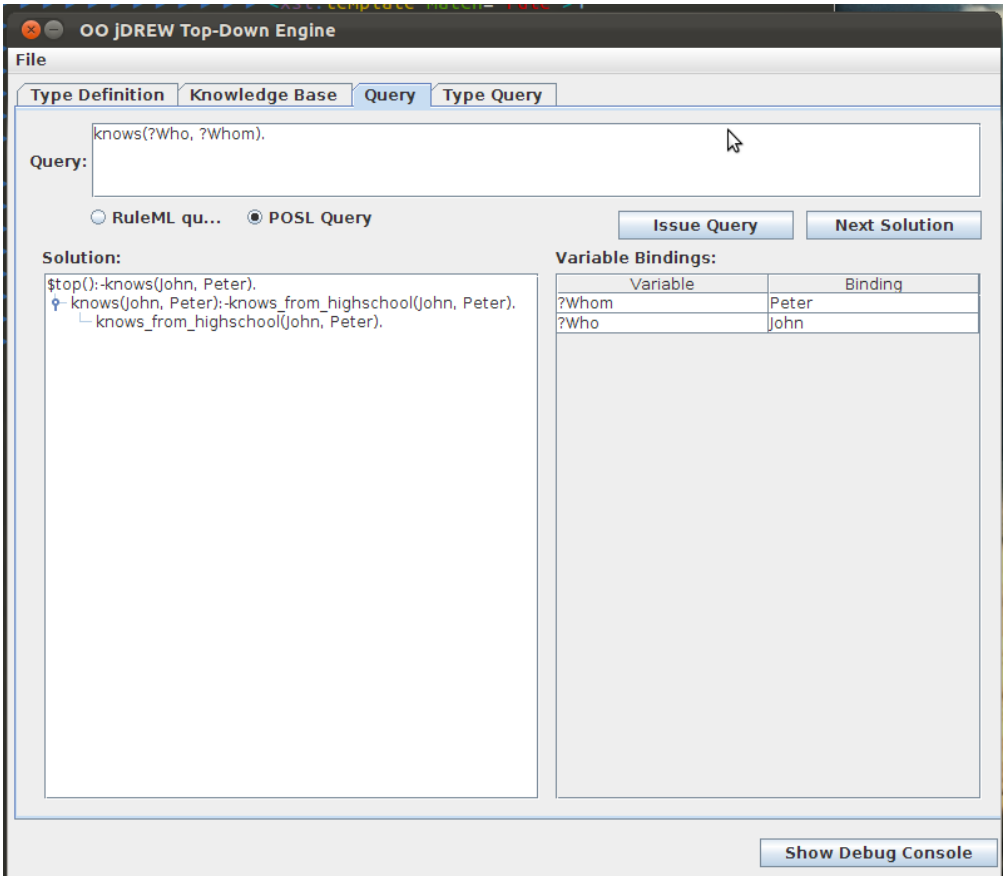
6.4.5 knows_trans(Mary, ?Whom)

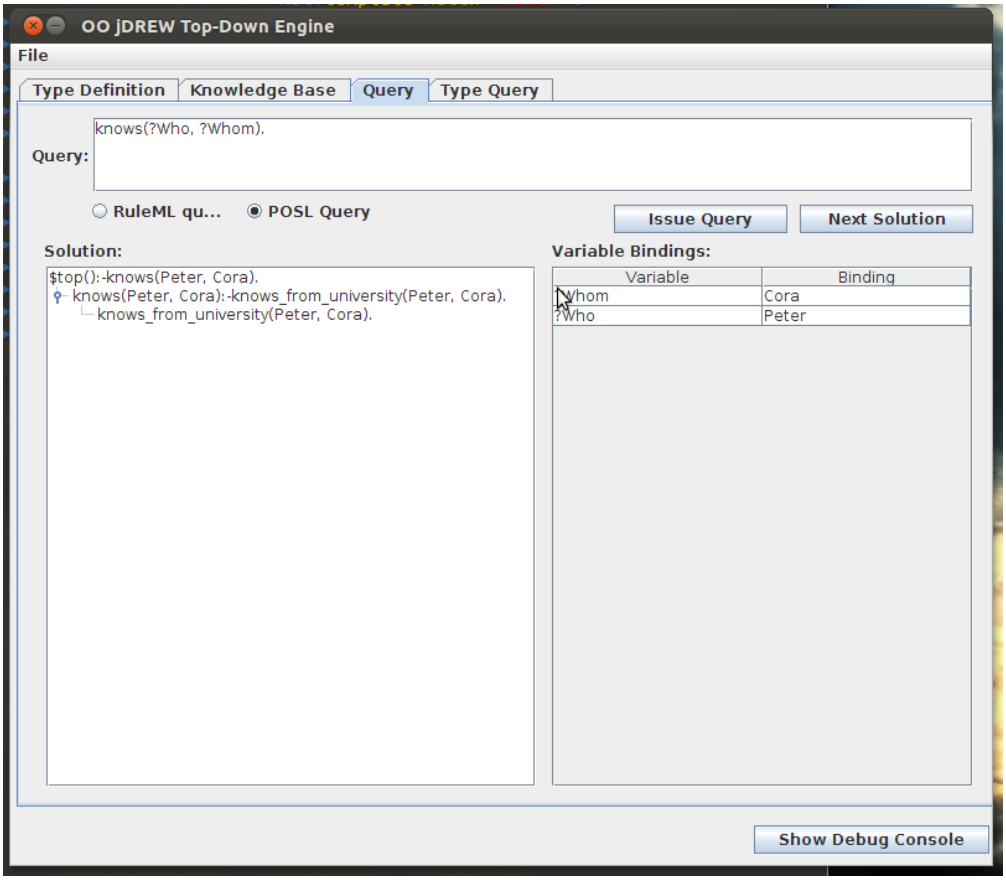
See following 4 screen shots.

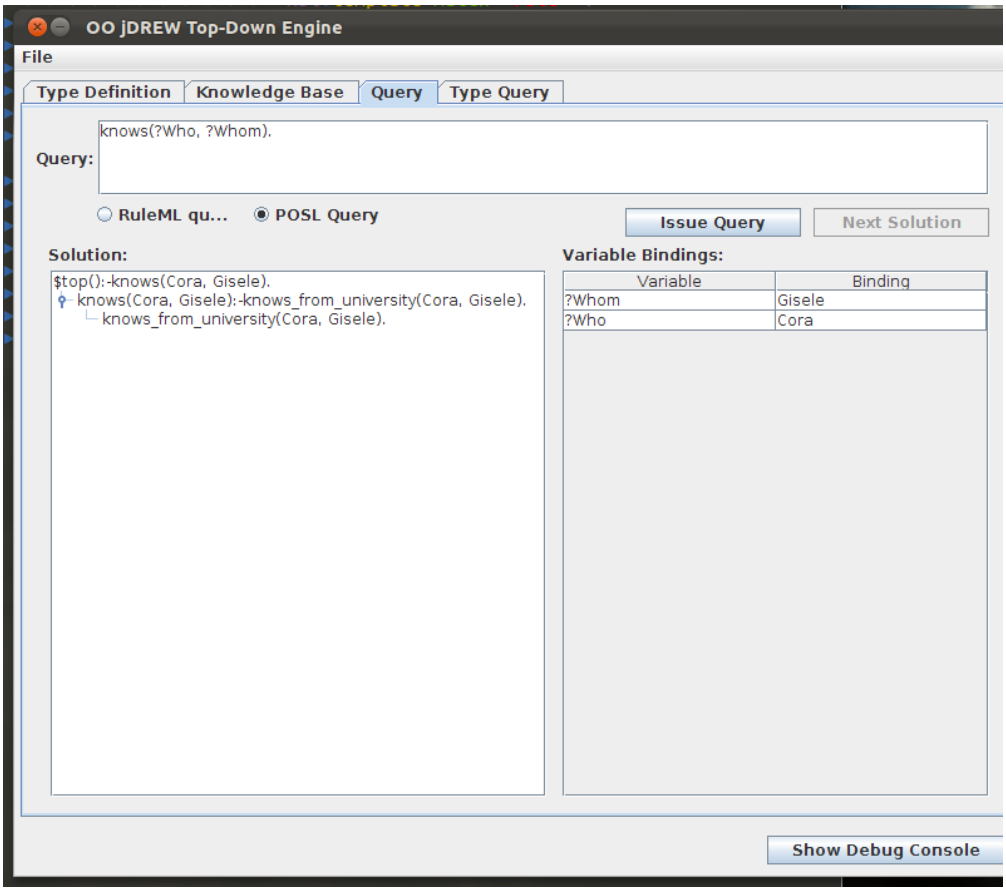
–

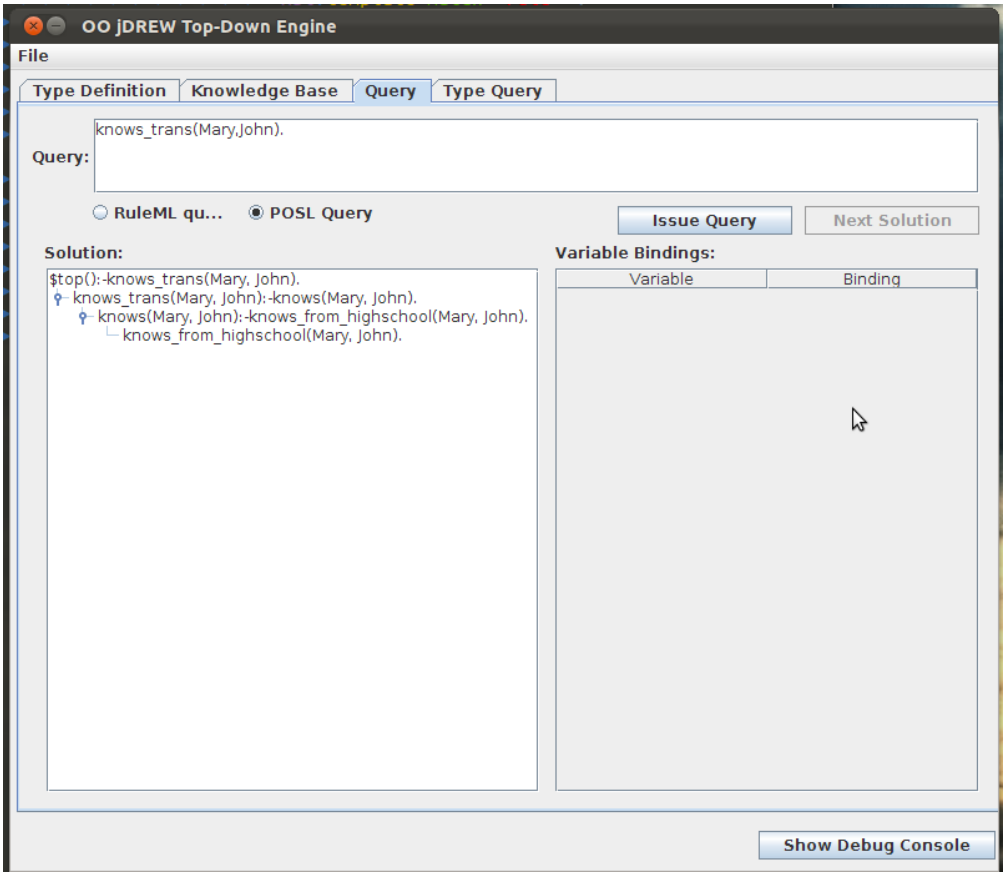
–

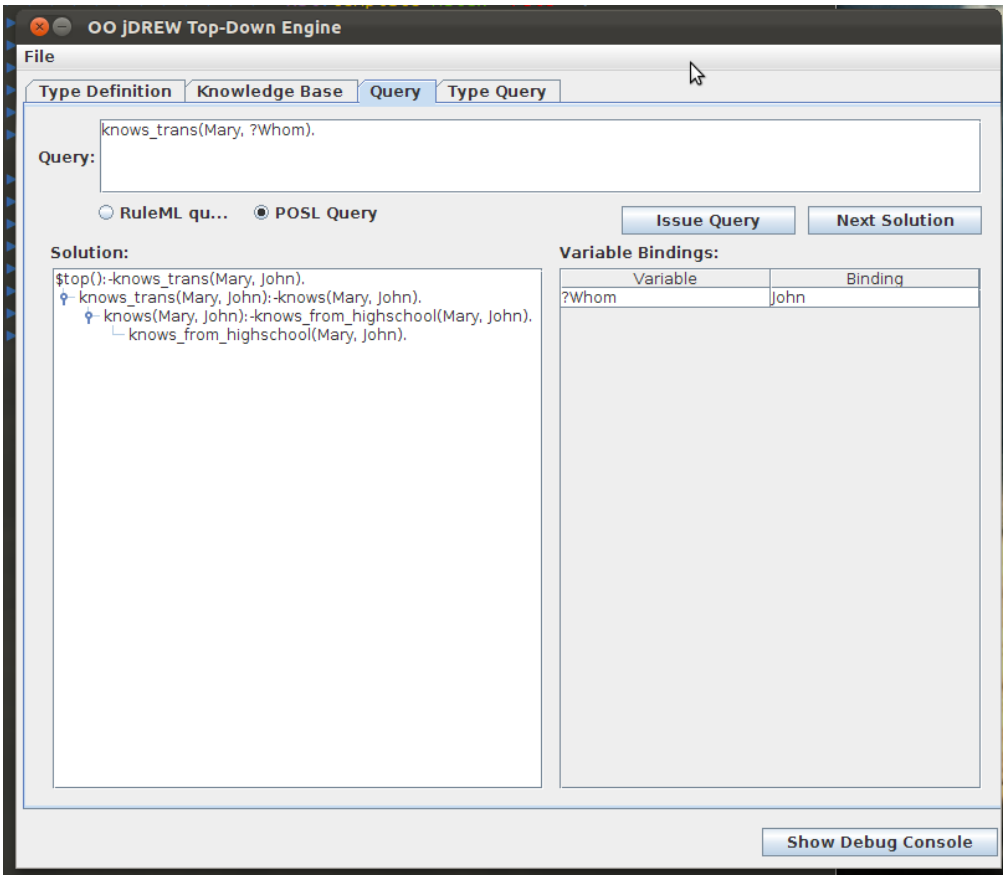
–

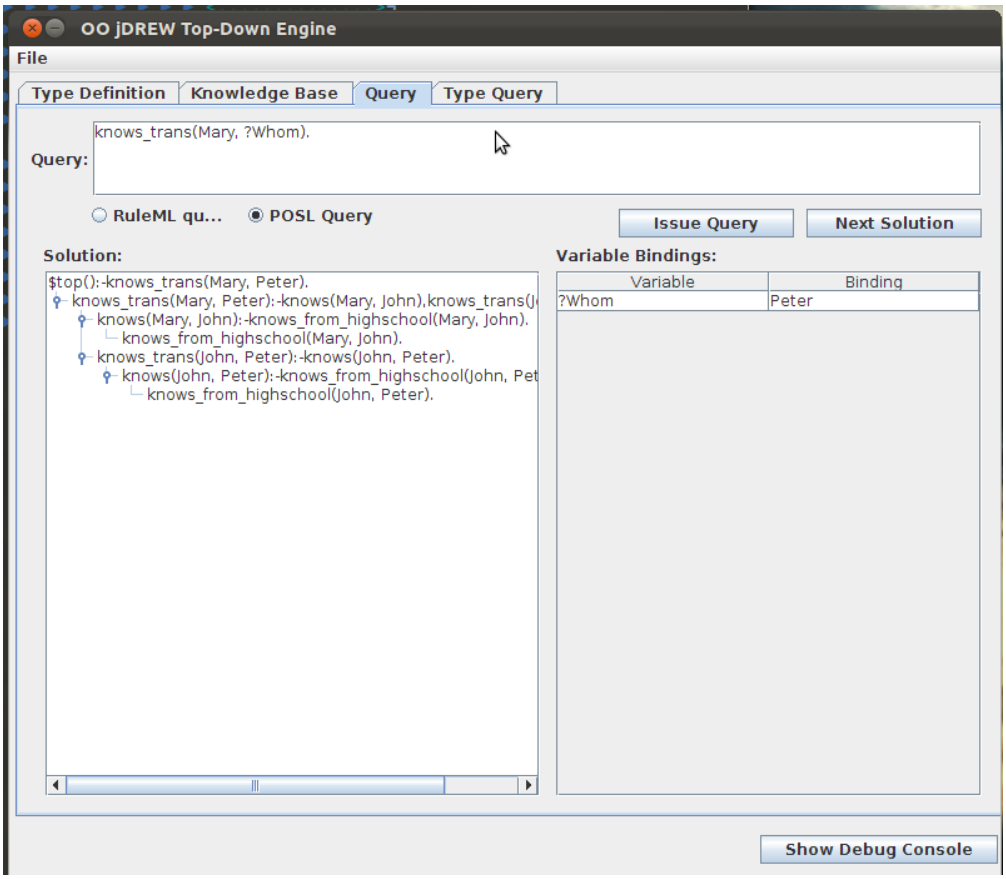

OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows(?Who, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows(John, Peter).
knows(John, Peter):-knows_from_highschool(John, Peter).
knows_from_highschool(John, Peter).
Variable Bindings:
Variable
Binding
?Whom
Peter
?Who
John
Show Debug Console

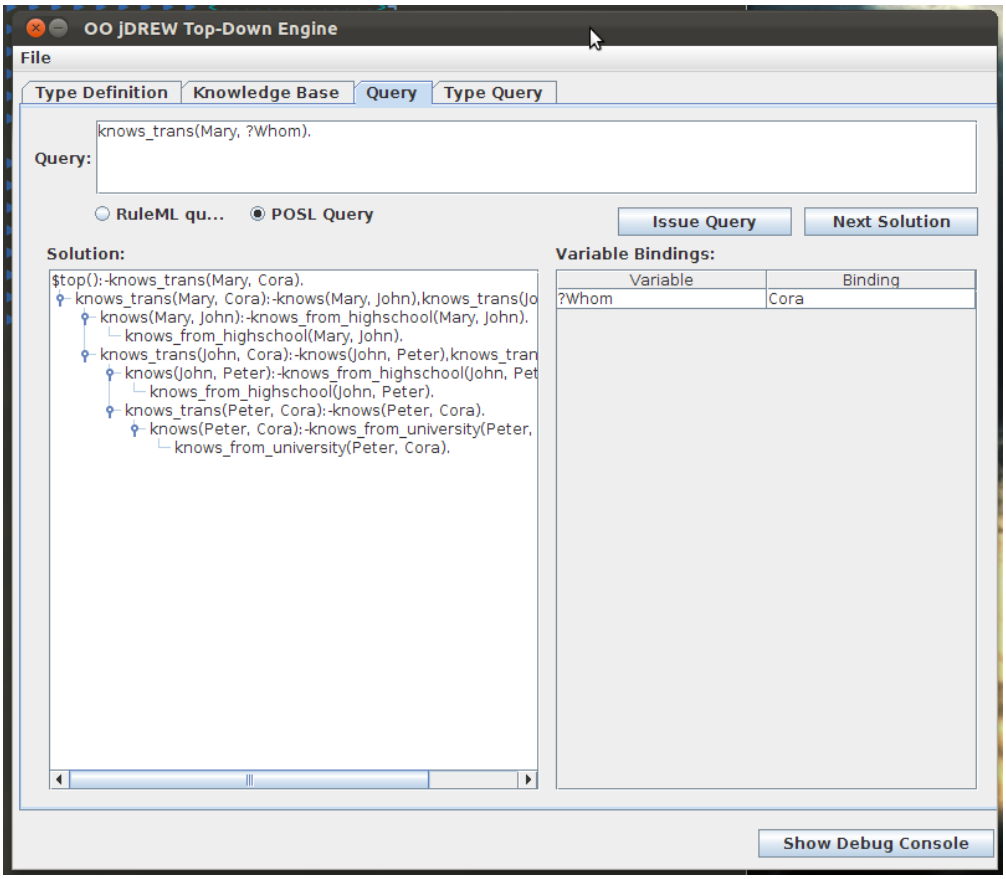

OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows(?Who, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows(Peter, Cora).
knows(Peter, Cora):-knows_from_university(Peter, Cora).
knows_from_university(Peter, Cora).
Variable Bindings:
Variable
Binding
?Whom
Cora
?Who
Peter
Show Debug Console

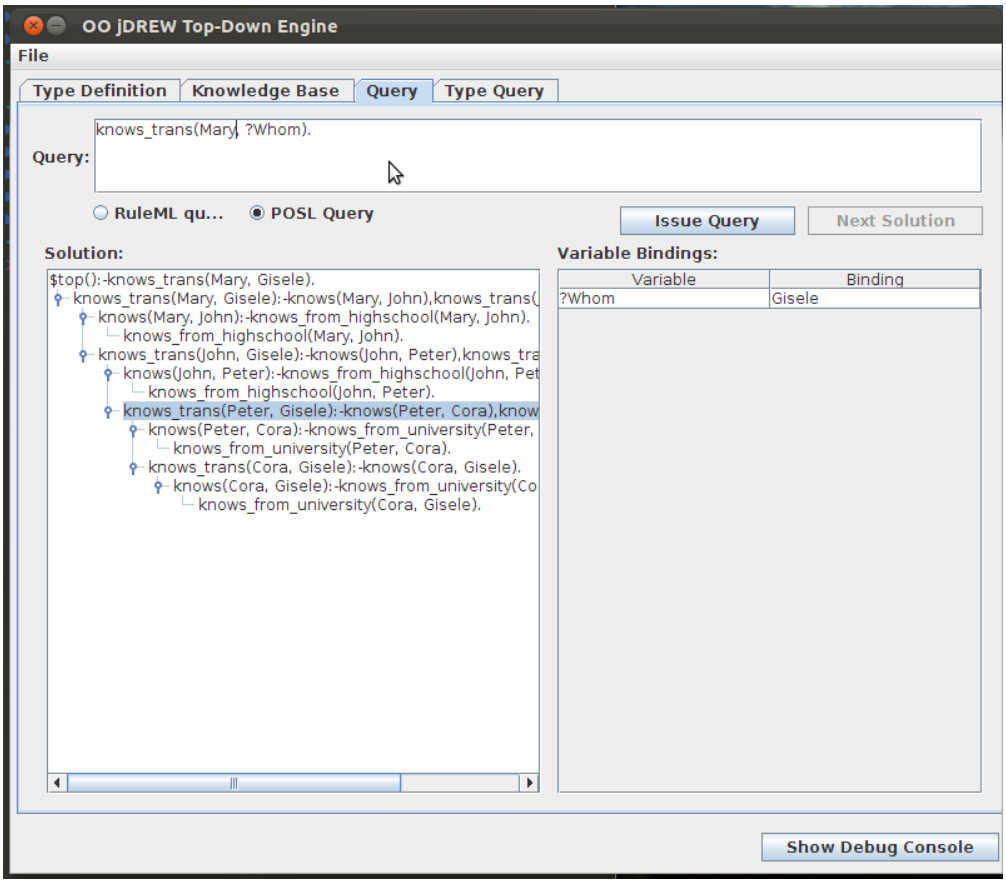

OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows(?Who, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows(Cora, Gisele).
knows(Cora, Gisele):-knows_from_university(Cora, Gisele).
knows_from_university(Cora, Gisele).
Variable Bindings:
Variable
Binding
?Whom
Gisele
?Who
Cora
Show Debug Console


OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows_trans(Mary,John).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows_trans(Mary, John).
knows_trans(Mary, John):-knows(Mary, John).
knows(Mary, John):-knows_from_highschool(Mary, John).
knows_from_highschool(Mary, John).
Variable Bindings:
Variable
Binding
Show Debug Console


OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows_trans(Mary, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows_trans(Mary, John).
knows_trans(Mary, John):-knows(Mary, John).
knows(Mary, John):-knows_from_highschool(Mary, John).
knows_from_highschool(Mary, John).
Variable Bindings:
Variable
Binding
?Whom
John
Show Debug Console

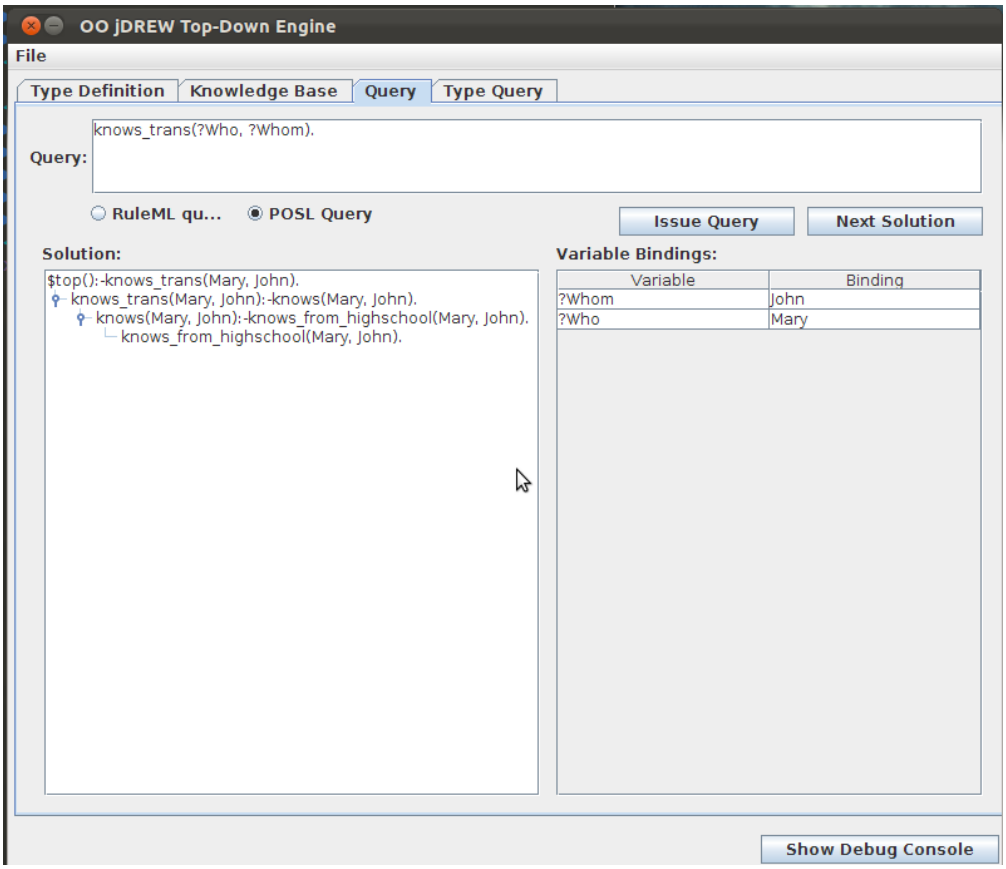

OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows_trans(Mary, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows_trans(Mary, Peter).
knows_trans(Mary, Peter):-knows(Mary, John),knows_trans(J
knows(Mary, John):-knows_from_highschool(Mary, John).
knows_from_highschool(Mary, John).
knows_trans(John, Peter):-knows(John, Peter).
knows(John, Peter):-knows_from_highschool(John, Pet
knows_from_highschool(John, Peter).
Variable Bindings:
Variable
Binding
?Whom
Peter
Show Debug Console


OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows_trans(Mary, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows_trans(Mary, Cora).
knows_trans(Mary, Cora):-knows(Mary, John),knows_trans(Jo
knows(Mary, John):-knows_from_highschool(Mary, John).
knows_from_highschool(Mary, John).
knows_trans(John, Cora):-knows(John, Peter),knows_tran
knows(John, Peter):-knows_from_highschool(John, Pet
knows_from_highschool(John, Peter).
knows_trans(Peter, Cora):-knows(Peter, Cora).
knows(Peter, Cora):-knows_from_university(Peter,
knows_from_university(Peter, Cora).
Variable Bindings:
Variable
Binding
?Whom
Cora
Show Debug Console


OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows_trans(Mary, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows_trans(Mary, Gisele).
knows_trans(Mary, Gisele):-knows(Mary, John),knows_trans(
knows(Mary, John):-knows_from_highschool(Mary, John).
knows_from_highschool(Mary, John).
knows_trans(John, Gisele):-knows(John, Peter),knows_tra
knows(John, Peter):-knows_from_highschool(John, Pet
knows_from_highschool(John, Peter).
knows_trans(Peter, Gisele):-knows(Peter, Cora),know
knows(Peter, Cora):-knows_from_university(Peter,
knows_from_university(Peter, Cora).
knows_trans(Cora, Gisele):-knows(Cora, Gisele).
knows(Cora, Gisele):-knows_from_university(Co
knows_from_university(Cora, Gisele).
Variable Bindings:
Variable
Binding
?Whom
Gisele
Show Debug Console

6.4.6 knows_trans(?Who, ?Whom)

See following 10 Screen shots.



–

–

—

–

–

–

—

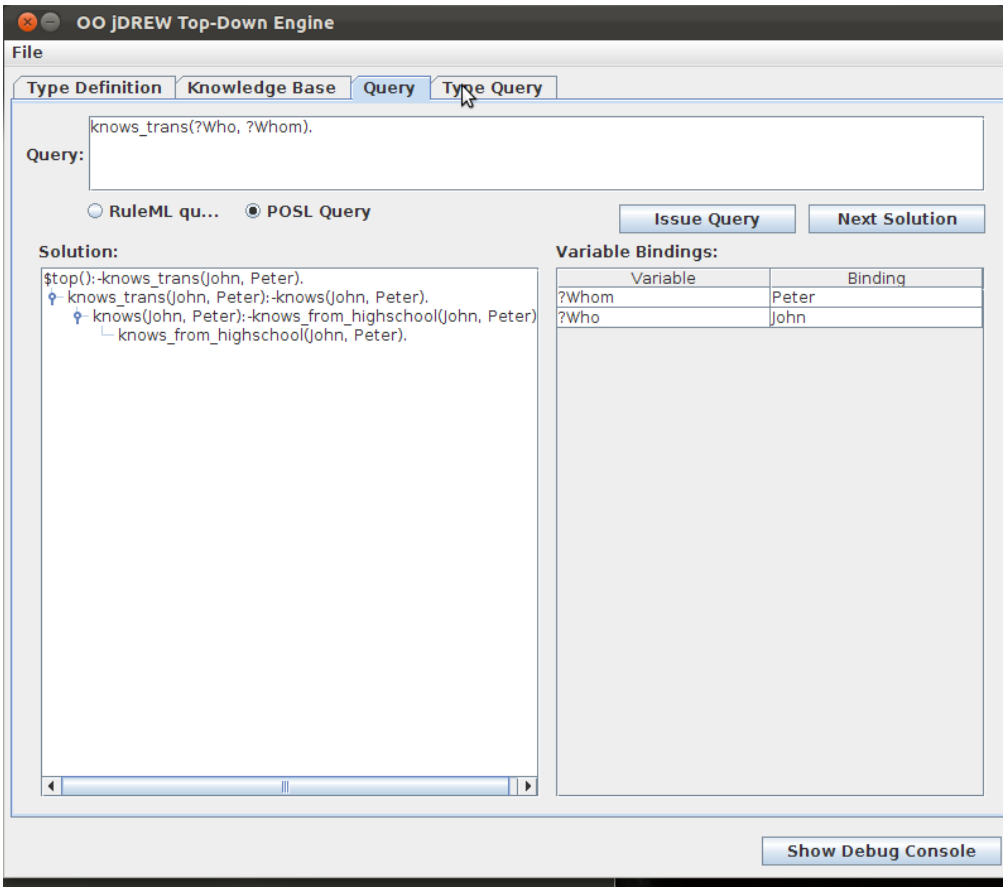
–

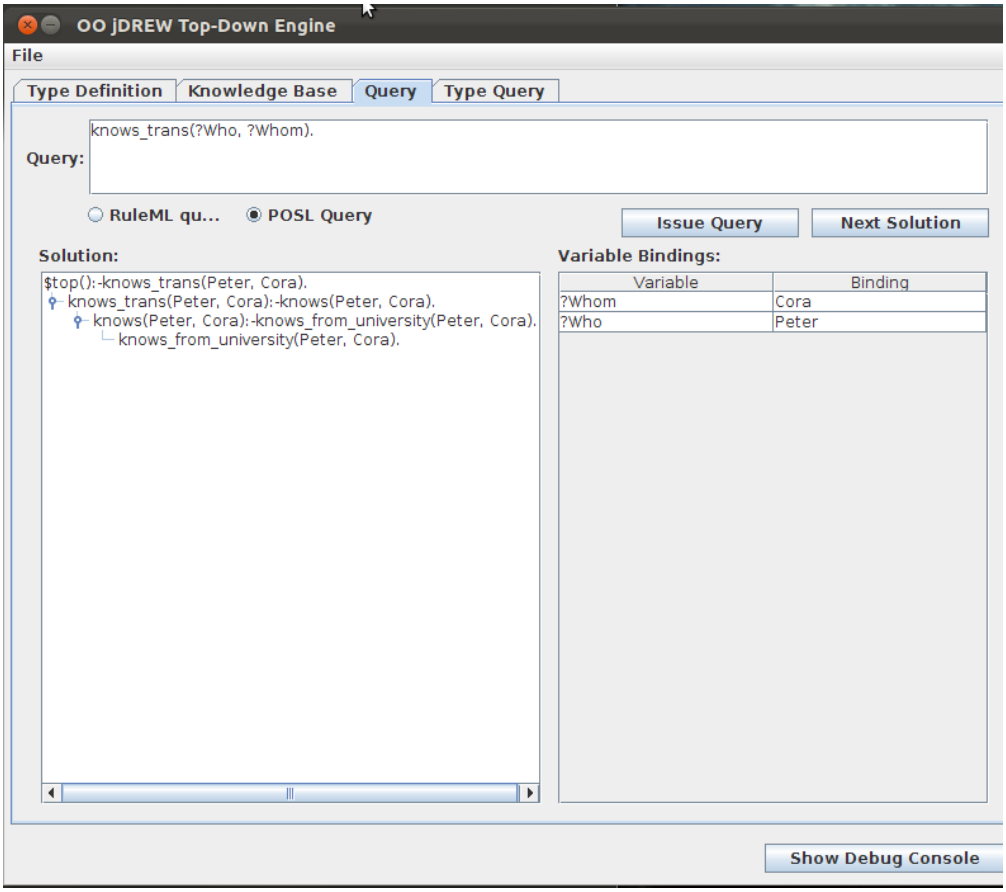
–

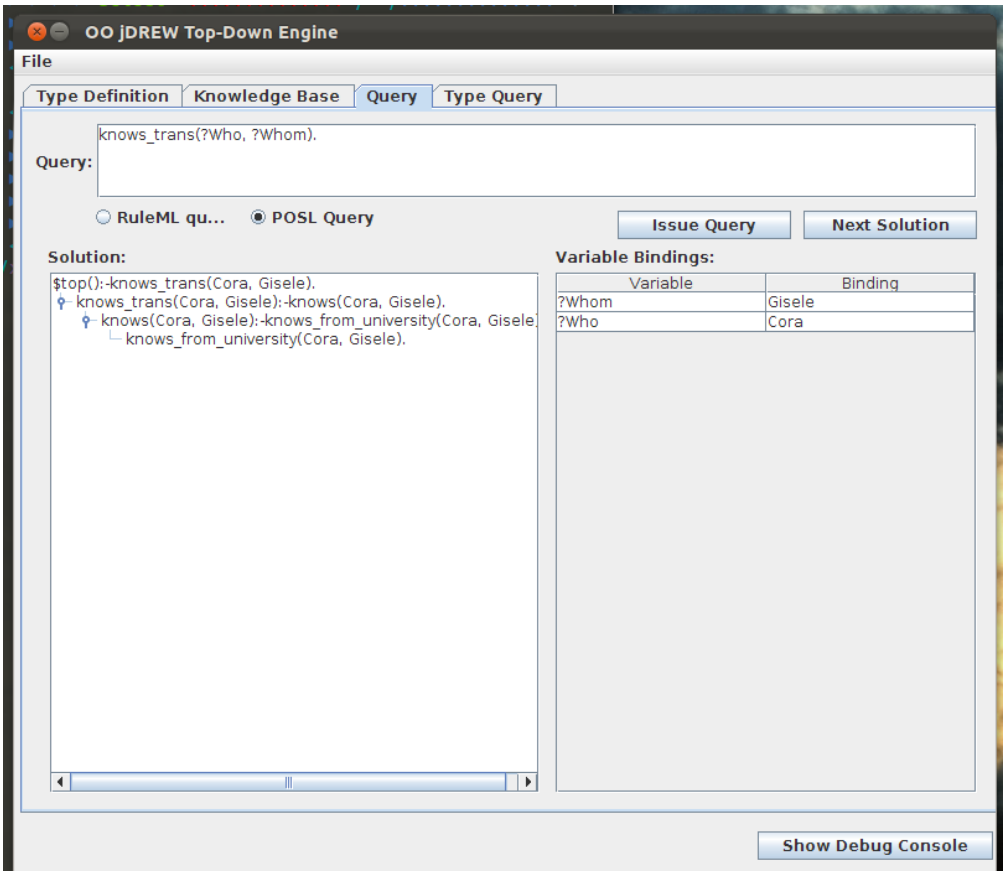
Todo

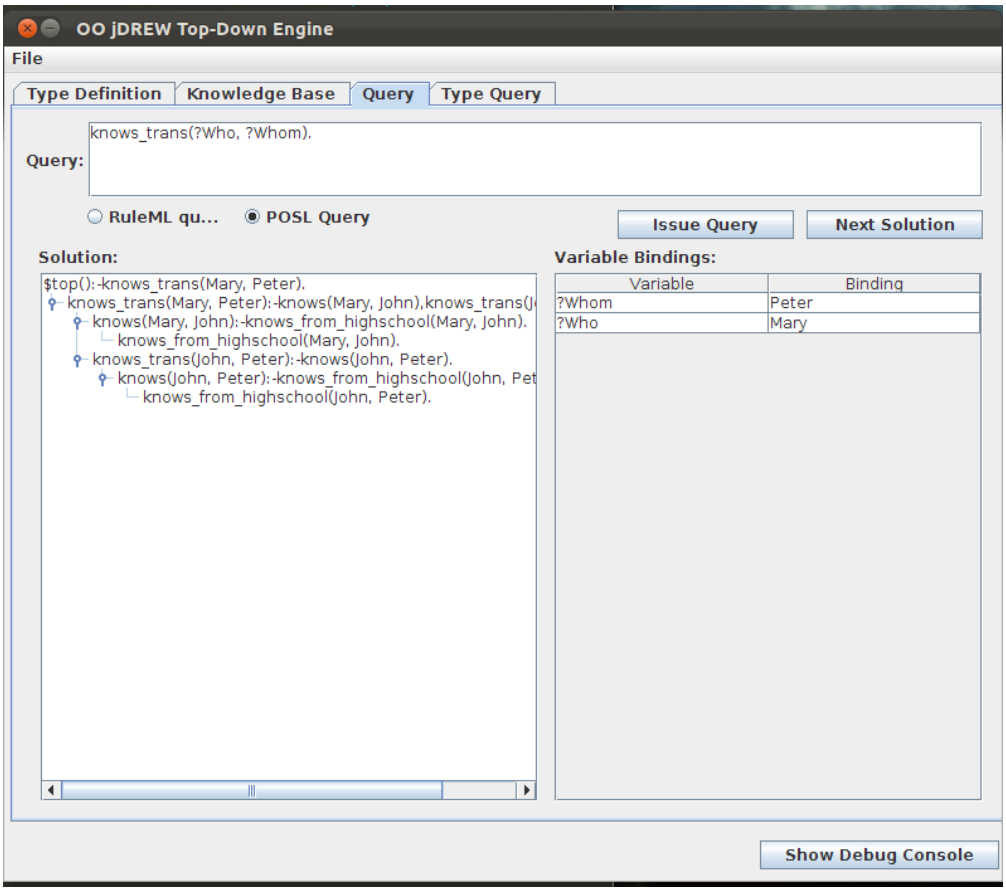
Give all results of the (bottom-up) generation employing OO jDREW BU:

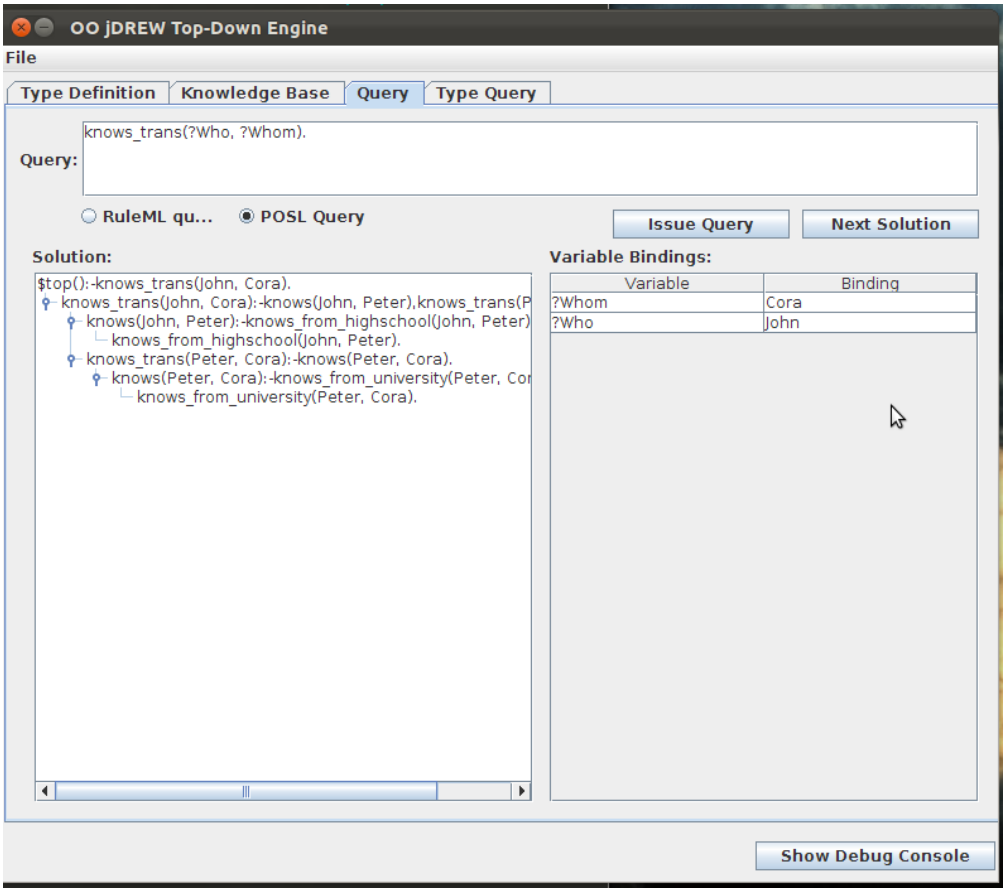
6.4.7 Top-down and Bottom-up correspondence

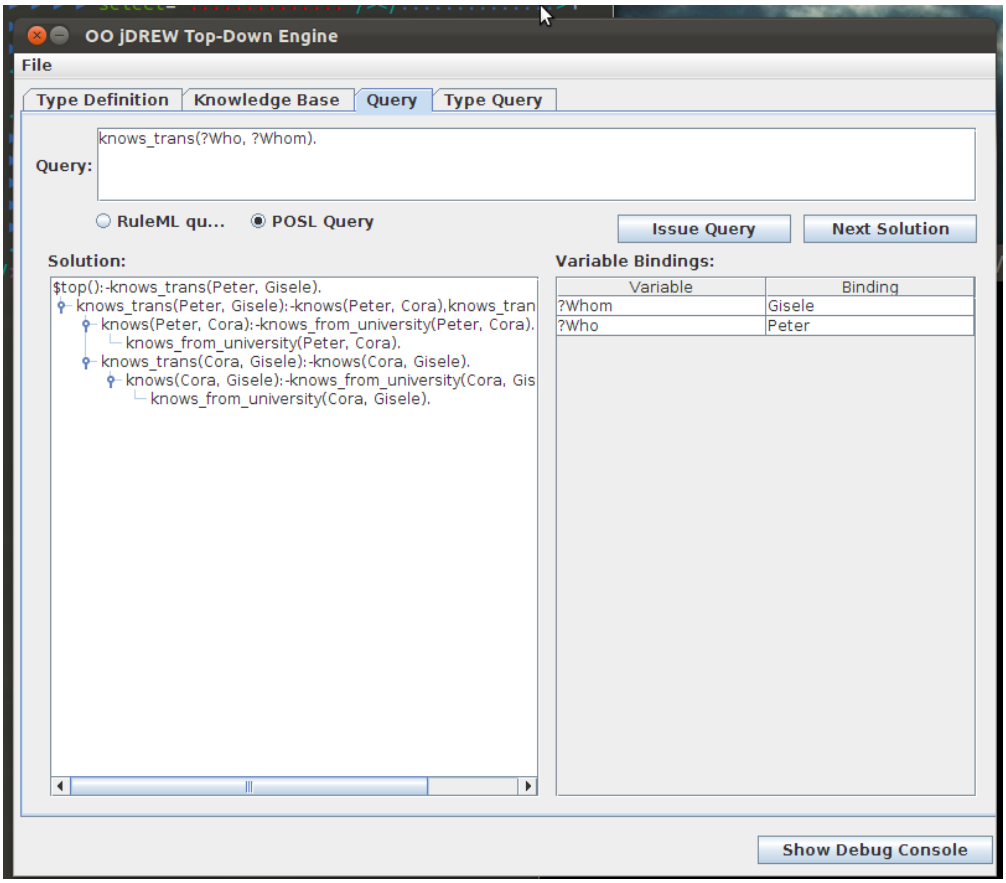

OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows_trans(?Who, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows_trans(John, Peter).
knows_trans(John, Peter):-knows(John, Peter).
knows(John, Peter):-knows_from_highschool(John, Peter)
knows_from_highschool(John, Peter).
Variable Bindings:
Variable
Binding
?Whom
Peter
?Who
John
Show Debug Console

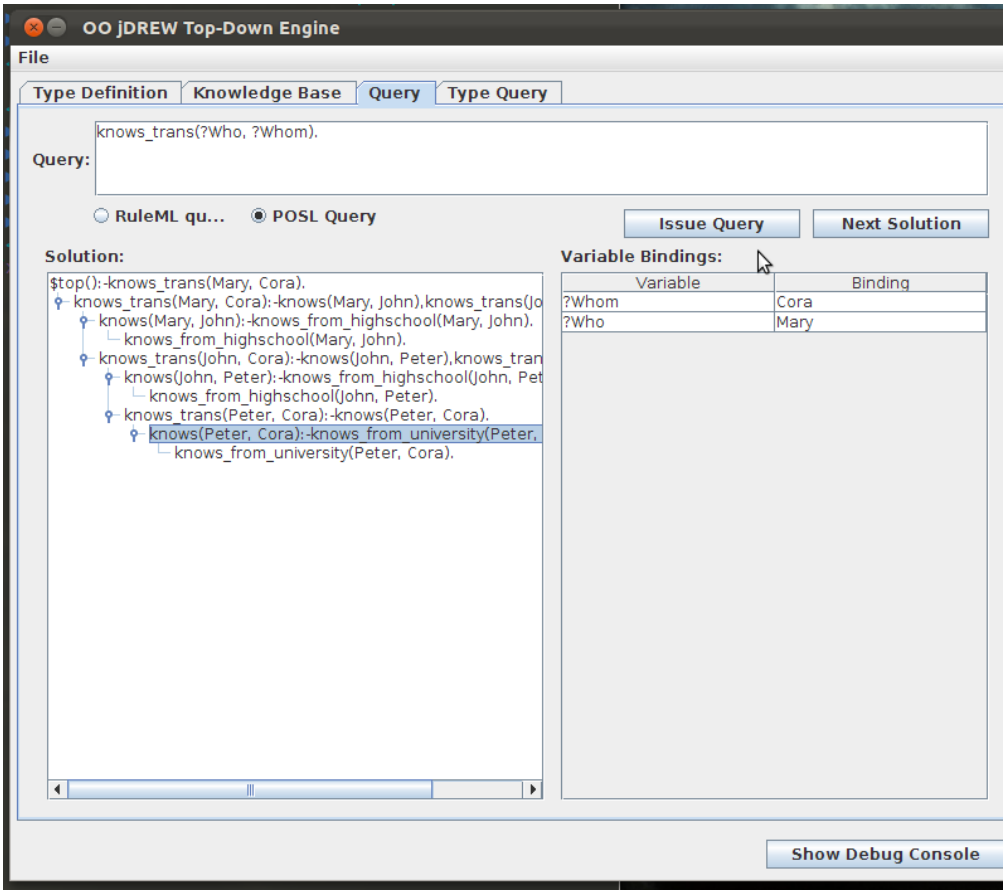

OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows_trans(?Who, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows_trans(Peter, Cora).
knows_trans(Peter, Cora):-knows(Peter, Cora).
knows(Peter, Cora):-knows_from_university(Peter, Cora).
knows_from_university(Peter, Cora).
Variable Bindings:
Variable
Binding
?Whom
Cora
?Who
Peter
Show Debug Console

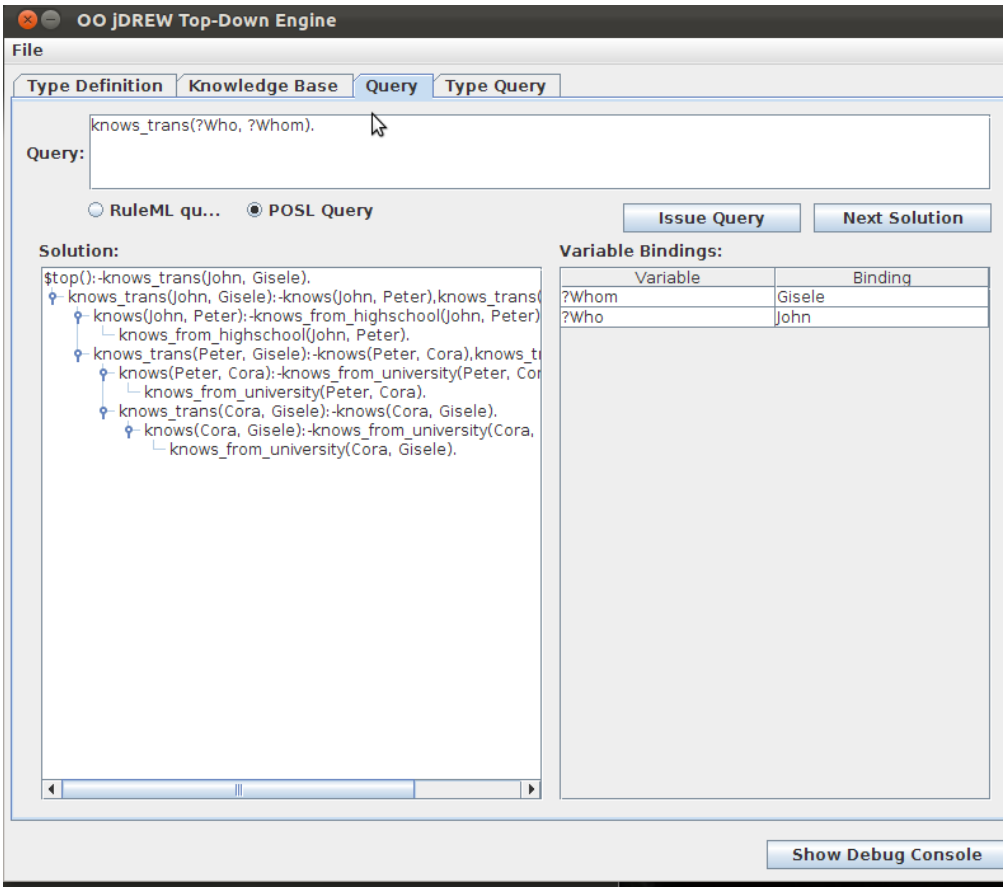

OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows_trans(?Who, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows_trans(Cora, Gisele).
knows_trans(Cora, Gisele):-knows(Cora, Gisele).
knows(Cora, Gisele):-knows_from_university(Cora, Gisele).
knows_from_university(Cora, Gisele).
Variable Bindings:
Variable
Binding
?Whom
Gisele
?Who
Cora
Show Debug Console

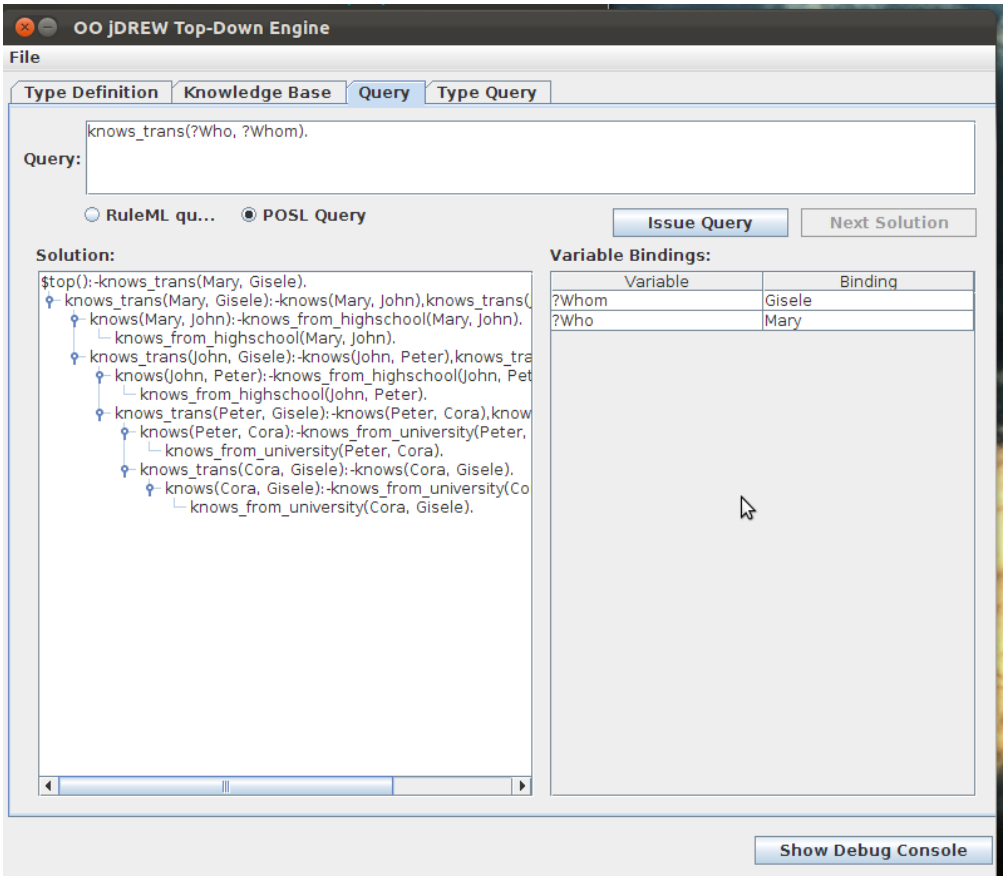

OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows_trans(?Who, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows_trans(Mary, Peter).
knows_trans(Mary, Peter):-knows(Mary, John),knows_trans(J
knows(Mary, John):-knows_from_highschool(Mary, John).
knows_from_highschool(Mary, John).
knows_trans(John, Peter):-knows(John, Peter).
knows(John, Peter):-knows_from_highschool(John, Pet
knows_from_highschool(John, Peter).
Variable Bindings:
Variable
Binding
?Whom
Peter
?Who
Mary
Show Debug Console

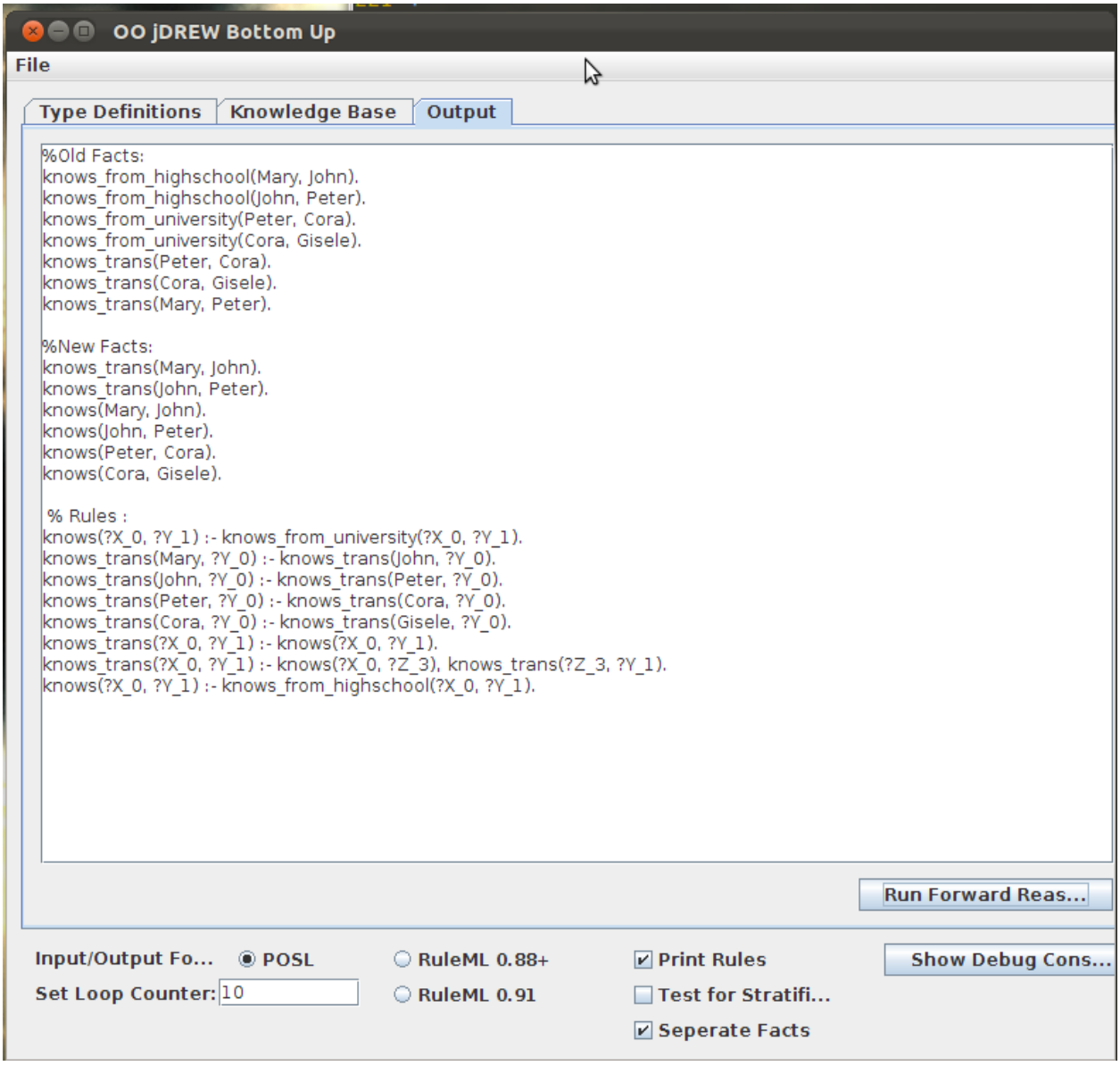

OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows_trans(?Who, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows_trans(John, Cora).
knows_trans(John, Cora):-knows(John, Peter),knows_trans(P
knows(John, Peter):-knows_from_highschool(John, Peter)
knows_from_highschool(John, Peter).
knows_trans(Peter, Cora):-knows(Peter, Cora).
knows(Peter, Cora):-knows_from_university(Peter, Cor
knows_from_university(Peter, Cora).
Variable Bindings:
Variable
Binding
?Whom
Cora
?Who
John
Show Debug Console


OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows_trans(?Who, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows_trans(Peter, Gisele).
knows_trans(Peter, Gisele):-knows(Peter, Cora),knows_tran
knows(Peter, Cora):-knows_from_university(Peter, Cora).
knows_from_university(Peter, Cora).
knows_trans(Cora, Gisele):-knows(Cora, Gisele).
knows(Cora, Gisele):-knows_from_university(Cora, Gis
knows_from_university(Cora, Gisele).
Variable Bindings:
Variable
Binding
?Whom
Gisele
?Who
Peter
Show Debug Console


OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows_trans(?Who, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows_trans(Mary, Cora).
knows_trans(Mary, Cora):-knows(Mary, John),knows_trans(Jo
knows(Mary, John):-knows_from_highschool(Mary, John).
knows_from_highschool(Mary, John).
knows_trans(John, Cora):-knows(John, Peter),knows_tran
knows(John, Peter):-knows_from_highschool(John, Pet
knows_from_highschool(John, Peter).
knows_trans(Peter, Cora):-knows(Peter, Cora).
knows(Peter, Cora):-knows_from_university(Peter,
knows_from_university(Peter, Cora).
Variable Bindings:
Variable
Binding
?Whom
Cora
?Who
Mary
Show Debug Console


OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows_trans(?Who, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows_trans(John, Gisele).
knows_trans(John, Gisele):-knows(John, Peter),knows_trans(
knows(John, Peter):-knows_from_highschool(John, Peter)
knows_from_highschool(John, Peter).
knows_trans(Peter, Gisele):-knows(Peter, Cora),knows_t
knows(Peter, Cora):-knows_from_university(Peter, Cor
knows_from_university(Peter, Cora).
knows_trans(Cora, Gisele):-knows(Cora, Gisele).
knows(Cora, Gisele):-knows_from_university(Cora,
knows_from_university(Cora, Gisele).
Variable Bindings:
Variable
Binding
?Whom
Gisele
?Who
John
Show Debug Console


OO jDREW Top-Down Engine
File
Type Definition
Knowledge Base
Query
Type Query
Query:
knows_trans(?Who, ?Whom).
RuleML qu...
POSL Query
Issue Query
Next Solution
Solution:
$top():-knows_trans(Mary, Gisele).
knows_trans(Mary, Gisele):-knows(Mary, John),knows_trans(
knows(Mary, John):-knows_from_highschool(Mary, John).
knows_from_highschool(Mary, John).
knows_trans(John, Gisele):-knows(John, Peter),knows_tra
knows(John, Peter):-knows_from_highschool(John, Pet
knows_from_highschool(John, Peter).
knows_trans(Peter, Gisele):-knows(Peter, Cora),know
knows(Peter, Cora):-knows_from_university(Peter,
knows_from_university(Peter, Cora).
knows_trans(Cora, Gisele):-knows(Cora, Gisele).
knows(Cora, Gisele):-knows_from_university(Co
knows_from_university(Cora, Gisele).
Variable Bindings:
Variable
Binding
?Whom
Gisele
?Who
Mary
Show Debug Console


OO jDREW Bottom Up
File
Type Definitions
Knowledge Base
Output
%Old Facts:
knows_from_highschool(Mary, John).
knows_from_highschool(John, Peter).
knows_from_university(Peter, Cora).
knows_from_university(Cora, Gisele).
knows_trans(Peter, Cora).
knows_trans(Cora, Gisele).
knows_trans(Mary, Peter).
%New Facts:
knows_trans(Mary, John).
knows_trans(John, Peter).
knows(Mary, John).
knows(John, Peter).
knows(Peter, Cora).
knows(Cora, Gisele).
% Rules :
knows(?X_0, ?Y_1) :- knows_from_university(?X_0, ?Y_1).
knows_trans(Mary, ?Y_0) :- knows_trans(John, ?Y_0).
knows_trans(John, ?Y_0) :- knows_trans(Peter, ?Y_0).
knows_trans(Peter, ?Y_0) :- knows_trans(Cora, ?Y_0).
knows_trans(Cora, ?Y_0) :- knows_trans(Gisele, ?Y_0).
knows_trans(?X_0, ?Y_1) :- knows(?X_0, ?Y_1).
knows_trans(?X_0, ?Y_1) :- knows(?X_0, ?Z_3), knows_trans(?Z_3, ?Y_1).
knows(?X_0, ?Y_1) :- knows_from_highschool(?X_0, ?Y_1).
Run Forward Reas...
Input/Output Fo...
POSL
RuleML 0.88+
RuleML 0.91
Print Rules
Test for Stratifi...
Seperate Facts
Show Debug Cons...
Set Loop Counter:
10

Todo

To which (top-down) query does the (bottom-up) generation correspond?

```
knows_trans(?Who,?Whom).
```

Todo

Briefly explain this correspondence.

Since bottom-up generation involves iteratively/recursively finding all grounded atoms, i.e. the least Herbrand model (`M`), the `knows_trans(X,Y).` transitive query will essentially solve the `M` if allowed to search until all solutions are found.

6.5 Part 5

Todo

Construct a small ontology with a class Public Transport that has four indirect subclasses, Bus, Streetcar, Metro, and Train. Consider Bus and Streetcar value restriction properties "borne Street"; Streetcar, Metro, and Train value restriction properties "borne Rail"; a Metro exists restriction property "level Subsurface"; for all four classes, value restriction properties "carry Person". Introduce two intermediate classes which abstract shared property restriction classes, give them (meaningful) names, and add their subclass relationships. Introduce all property restriction classes at the highest possible levels. Introduce Metro instances m1 and m2, Train instance t, and Person instance p. Represent the facts that m1 and m2 carry p, and t carries p.

Write all property restriction classes that can be derived for subclasses, here:

- borneStreet all StreetTransport
- borneRail all RailTransport
- levelSubsurface some Metro
- carryPerson some Public_transport

6.5.1 Part 5 A

Draw a diagram for the ontology. Hint: Plan to best use the space below. Hint: Alternatively, you can model everything in the Protégé ontology editor and get the diagram from its Jambalaya tab (attach a printout).

6.5.2 Part 5 B

ABox

{m1:Metro, m2:Metro, t:Train, p:Person}

TBox

(see attached sheet.)

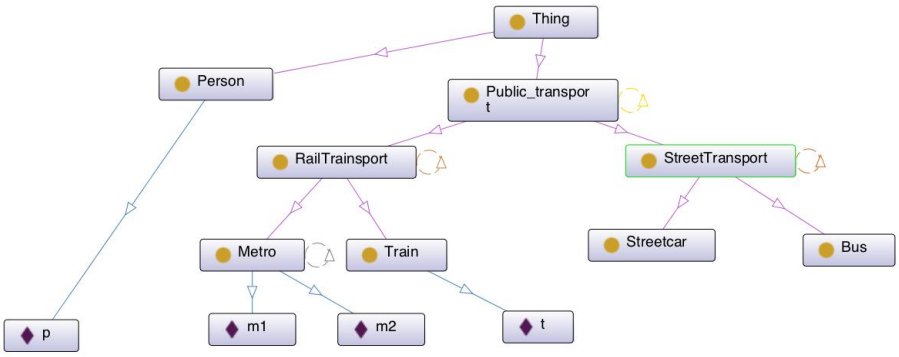


Figure 6.2: Ontology diagram.

6.5.3 Part 5 C

There are no cases of *direct* multiple inheritance in my model. However, should I (for example) have abstracted `RailTransport` to the level of `Public_transport` then `Metro` and `Train` would inherit from both `Public_transport` and `RailTransport`.

However, there are cases of *inherited anonymous classes*, for example:

Metro and Train both inherit from `borneRail only RailTransport` and `carryPerson only Public_transport` (alos Metro inherits from `levelSubsurface some Metro`).

Bus and Streetcar inherit from `borneStreet only StreetTransport` and `carryPerson only Public_transport`.

`RailTransport` inherits from `Public_transport` and `borneRail only RailTransport`.

`StreetTransport` inherits from `Public_transport` and `borneStreet only StreetTransport`.

`Public_transport` inherits from `carryPerson only Public_transport` and `Thing`.

The reasoning tasks performed on the created ontology was the Pellet and Hermit plugins for the Protege Owl 4 framework. *(There did not appear to be anything new learned by starting the mentioned reasoner. Protege is a very complicated tool and the interface is unintuitive, which is also reflected in the documentation.)*

However the key reasoning tasks done by the above reasoners based on the tableaux algorithm (namely Pellet) include:

- Satisfiability
- Instance checking
- Concept satisfiability
- Retrieval
- Concept Subsumption
- and Equivalence

BIBLIOGRAPHY

[JSON] JSON the Fat Free Alternative to XML, Introducing JSON, 21 September 2011, <http://www.json.org/xml.html>.

[Tolf11] Why XML is bad for representing arbitrary data, Home Page of Fredrik Tolf, 21 September 2011, <http://dolda2000.com/~fredrik/doc/xmlds>.

[Wp1] Wikipedia: Prolog, Wikipedia.org, 21 September 2011, <http://en.wikipedia.org/wiki/Prolog>.

[Brac04] Brachman R.J., and Levesque H.J., *Knowledge Representation and Reasoning*. San Francisco, CA: Elsevier, 2004.