

## CS 246 Exercise Set #2

In this assignment, you will explore different data formats and learn the basics of Infomaster. Your group will present your work (exercise #6 only) in class on Thursday, May 1. Please also email your answers to [mkassoff@cs](mailto:mkassoff@cs) by class time on Thursday (no hardcopy is required). You only need 1 copy of the written questions per group, but everyone should collaborate on the questions and understand the answers.

**Important note.** Currently, Infomaster uses classical negation as the semantics for negated queries. As discussed in class, this is not always desirable. You will likely want to use negation as failure semantics from time to time. To do so, you can use the procedural attachment **unprovable**. You use **unprovable** instead of **not** whenever you want negation as failure semantics. For example, you might want to say:

```
(<= (childless ?x)
    (person ?x)
    (unprovable (father ?x))
    (unprovable (mother ?x)))
```

When you want classical negation, use **not** as usual:

```
(<= (female ?x)
    (not (male ?x)))
```

*Another note:* For some reason, the Examine? command is not working properly in some browsers, for example IE for Mac OS X. Quotation marks (“”) are sometimes rendered as quot;, which is bad. You will get syntax errors due to this. If you encounter this problem, try using a different browser, i.e. Netscape 7.

### Exercise #5

For this exercise, you will augment your Advisor/Student database from Exercise Set #1 with rules.

- 0) **Extra credit:** For those groups that did not get perfect scores on Exercise 2 on the first assignment, fix your database according to my written comments and the answers to the first assignment on the website. You can get back up to 75% of the points that you lost on the first assignment on Exercise 2 (I’ll take the points you earned back multiplied by .75 and add that to your first assignment score). Please write down what you have fixed and how many points you lost for it before so that I can do the grading (I don’t have detailed records of how every group lost points on the first assignment).
- 1) (5 points) Give each object in your system a primary class. This is done by adding “isa” facts to the system. For instance, to give Stanford the primary class Institution, add the fact (isa stanford institution) to the Repository. You can either

use the Assert or Examine commands to add this information to the system (see step 2). Note that now you can view your objects with the Inspect command and they will be viewed as being a member of their primary class. Note that, of course, each object should have exactly one primary class.

- 2) (5 points) Write query rules to automate class inheritance (i.e., all Advisors are be Persons and should have all attributes of Persons). Note that you can use the command <http://localhost:3000/repository/assert?> to enter rules. You can also use the retract button to retract rules, but it is probably easier to use the Examine command to edit the rules. Actually you may find it easier to use Examine to enter rules as well, but this is up to you. Use Examine to remove unnecessary stored links from your system. For example, (Person.Instance somestudent) is not necessary if (Student.Instance somestudent) plus your new rules are in the system. Remember, you want to use rules with the backwards arrow, not the forwards arrow. [Query rule: ( $\leftarrow$  (p ?x) (q ?x)) Update rule: ( $\Rightarrow$  (q ?x) (p ?x))]
- 3) (5 points) Write a query rule that allows Student.Advisor to be stored and Advisor.Student to not be stored. Write an update rules that allow users to update Advisor.Student. Make sure to handle both insertion and deletion. (Remember, concluding negated facts deletes them.) Remove redundant old stored data from your system. *Note:* When you update Advisor.Student and add a new student, that link will actually be stored in the Repository, since the Repository stores everything. Therefore you will be stuck with this new redundant data. As we will see in future lectures, there are ways to solve this problem, but they involve using other types of agents (not Fullservers).
- 4) (7 points) Create attributes Person.Firstname and Person.Lastname for the Person class. Using the Xform operation and the string manipulation procedural attachments, populate these fields from Person.Name. Remove the old Person.Name data from the system (the Examine command should be useful here) and remove the attribute itself from the system. Make sure to populate the new attributes with UI information, i.e. creation and search information. This goes for all attributes you create from now on.
- 5) (5 points) Create a forest (a set of trees) of students and advisors (similar to the Ullman tree), where one student has an advisor who was a student of his advisor, etc. Create a few different lineages, at least one with 3 generations and one with 4 generations. Make sure that these trees are indeed trees, not just linked lists. Note that a student should have exactly one advisor, not multiple advisors (like some of you allowed in the first exercise set). Also, if necessary change your metadata to allow advisors to have no students.
- 6) (5 points) Create an attribute Ancestor which belongs to both students and advisors. Write rules that allow Ancestor to be virtually populated that person's advisor, his advisor's advisor, etc. Since this attribute is virtual, you will want to

set its CreateStyle and ChangeStyle to Noshow. The same goes for Decendant and FamilyMember below.

- 7) (5 points) Create an attribute Decendant which belongs to advisors. Write rules that allow Decendant to be virtually populated with the students of that advisor, the student's students, etc.
- 8) (5 points) Create an attribute FamilyMember that is an attribute of both students and advisors. Write rules that allow FamilyMember to be virtually populated with all of the students and advisors in that person's tree (i.e. that person, that person's students and advisor, those people's students and advisors, etc.). *Written question 5.1:* Write down the rules from this exercise.
- 9) (20 points) Notice that the computation required by the rules in part 6 can be quite expensive, especially if the family is large (like Ullman's academic family). Every time some one inspects the member of a large family, a lot of CPU time is used and the response time of the system can be quite slow. To solve these problems, we will want to store, or materialize, the Family attribute. However note also that we don't want to store the Family attribute for every family member, as this is potentially a lot of redundant storage, not to mention a lot to be maintained when updates to the family occur. Therefore we will store the extension of the Family attribute only once for each family. Note that every family has one unique founder. For example, Jeff Ullman is the founder of the Ullman tree of students. You will store the Family data with each family's founder using update rules.

Create rules to deal with the addition of new members to a family, and the creation of a new family (i.e. a new advisor is created, who is the founder of his/her singleton family. You can assume that the new members are either (1) students of persons who are currently advisors, or (2) students of students, i.e. the student in the tree has become an advisor! In the second case, you should also automatically store the fact that the student who was not previously an advisor has become an advisor (but also remains a student). You can assume that only 1 person is added at once. Remove the rules you created in step 7 (which is why I asked you to write them down!) Replace these rules with rules that virtually populate every member of the tree's Family attribute (except, possibly, for the founder's Family attribute, which is of course stored).

- 10) **Extra credit:** (10 points) Write rules that allow you to merge two trees. Note that one founder will no longer be a founder, so this will require dematerializing the stored data for the previous founder and storing it with the other founding father.
- 11) **Dump** your database as exercise5.franz or exercise5.mcl. *Written question 5.2:* Indicate which parts of problems 7 and 8 you attempted. This will help me to know what to expect when I grade the assignment.

## **Exercise #6**

(30 points) Augment your database instance from Exercise 3 with rules. Add some new relations and attributes that will allow you to showcase some interesting rules during your presentation.