DatalogQ Interpreter

The DatalogQ interpreter is invoked using the following terminal command:

```
$ java DLOGQ company
```

Here `$` is the command prompt and `company` is the name of the database (as well as the name of the directory where the database files are stored). This command assumes that the `company` directory is present in the same directory where this command is issued. Of course, one can issue this command in a different directory by providing the full path to the database directory.

The interpreter responds with the following prompt:

```
DLGQ>
```

At this prompt the user may enter the query execution command `@file-name` or type the `exit` command, where file-name contains the DatalogQ query. Each command is to be terminated by a semi-colon. Even the `exit` command must end with a semi-colon.

Datalog Syntax

Datalog is a rule-based logical query language for relational databases. The syntax of Datalog is defined below:

An *atomic formula* is of one of the following two forms:

1. `p(x1, ..., xn)` where `p` is a relation name and `x1`, ..., `xn` are either constants or variables or
2. `x <op> y` where `x` and `y` are either constants or variables and `<op>` is one of the six comparison operators: `<, <=, >, >=, =, !=.`

A *Datalog rule* is of the form:

```
  p :- q1, ..., qn.
```

Here `p` is an atomic formula and `q1, ..., qn` are either atomic formulas or negated atomic formulas (i.e. atomic formula preceded by `not`). `p` is referred to as the head of the rule, and `q1, ..., qn` are referred to as sub-goals.

A Datalog rule `p :- q1, ..., qn`. is said to be *safe* if

1. Every variable that occurs in a negated sub-goal also appears in a positive sub-goal, and
2. Every variable that appears in the head of the rule also appears in the body of the rule.

A *Datalog query* is set of safe Datalog rules with at least one rule having the `answer` predicate in the head. The `answer` predicate collects all answers to the query.

Note: Variables that appear only once in a rule can be replaced by anonymous variables (represented by underscores). Every anonymous variable is different from all other variables.

Datalog Query Examples

The following are examples of Datalog queries against the company database:

Query 1: Get names of all employees in department 5 who work more than 10 hours/week on the ProductX project.

```
answer(F,M,L) :-
  employee(F,M,L,S,_,_,_,_,_,5),
  works_on(S,P,H),
  projects('ProductX',P,_,_),
  H >= 10.
```

Query 2: Get names of all employees who have a dependent with the same first name as their own first names.

```
answer(F,M,L) :-
  employee(F,M,L,S,_,_,_,_,_,_),
  dependent(S,F,_,_,_).
```

Query 3: Get the names of all employees who are directly supervised by Franklin Wong.

```
answer(F,M,L) :-
  employee(F,M,L,_,_,_,_,_,S,_),
  employee('Franklin',_,'Wong',S,_,_,_,_,_,_).
```

Query 4: Get the names of all employees who work on every project.

```
temp1(S,P) :-
  employee(_,_,_,S,_,_,_,_,_,_),
  projects(_,P,_,_).
temp2(S,P) :-
  works_on(S,P,_).
temp3(S) :-
  temp1(S,P), not temp2(S,P).
answer(F,M,L) :-
  employee(F,M,L,S,_,_,_,_,_,_), not temp3(S).
```

In this query, `temp1(S,P)` collects all combinations of employees, `S`, and projects, `P`;

`temp2(S,P)` collects only those pairs where employee `S` works on project `P`; `temp3(S)` collects employees, `S`, who do not work for a particular project (these employees should not be in the answer). A second negation in the final rule gets the answers to the query.

Query 5: Get the names of employees who do not work on any project.

```
  temp1(S) :-
    works_on(S,_,_).
  answer(F,M,L) :-
    employee(F,M,L,S,_,_,_,_,_,_), not temp1(S).
```

Query 6: Get the names and addresses of employees who work for at least one project located in Houston but whose department does not have a location in Houston.

```
  temp1(S) :-
    works_on(S,P,_), project(_,P,'Houston',_).
  temp2(S) :-
    employee(_,_,_,S,_,_,_,_,_,D),
    not dept_locations(D,'Houston').
  answer(F,M,L,A) :-
    employee(F,M,L,S,_,A,_,_,_,_), temp1(S), temp2(S).
```

`temp1(S)` collects employee `S` who work for a project located in `Houston`; `temp2(S)` collects employees `S` whose department do not have a location in `Houston`; the final rule intersects the two temp predicates to get the answer to the query.

Query 7: Get the names and addresses of employees who work for at least one project located in Houston or whose department does not have a location in Houston. (Note: this is a slight variation of the previous query with 'but' replaced by 'or').

```
  temp1(S) :-
    works_on(S,P,_),
    project(_,P,'Houston',_).
  temp2(S) :-
    employee(_,_,_,S,_,_,_,_,_,D),
    not dept_locations(D,'Houston').
  answer(F,M,L,A) :-
    employee(F,M,L,S,_,A,_,_,_,_), temp1(S).
  answer(F,M,L,A) :-
    employee(F,M,L,S,_,A,_,_,_,_), temp2(S).
```

Query 8: Get the last names of all department managers who have no dependents.

```
temp1(S) :-
  dependent(S,_,_,_,_).
answer(L) :-
```

```
  employee(_,_,L,S,_,_,_,_,_,_),
  department(_,_,S,_),
  not temp1(S).
```

To execute the above queries using the Datalog interpreter, each must be placed in a separate file with a `$` symbol appearing at the end of the file. Assume that the queries are placed in files named `q1, q2, …, q8`. The following is a terminal session showing the execution of the above queries:

```
[raj@tinman ch2]$ java DLOGQ company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q1;
------------------------------------
answer(F,M,L) :-
  employee(F,M,L,S,_,_,_,_,_,5),
  works_on(S,P,H), H >= 10,
  projects('ProductX',P,_,_).$
------------------------------------
ANSWER(F:VARCHAR,M:VARCHAR,L:VARCHAR)

Number of tuples = 2
John:B:Smith:
Joyce:A:English:

DLOG> exit;
Exiting...
```

DatalogQ Syntax

DatalogQ is an extension of Datalog that allows for universally-quantified conditions to be introduced in the body of rules using “complex” terms.

Complex Terms:

In addition to constants and variables that are available in Datalog, DatalogQ allows complex terms of the form:

```
[*]:p(t1,…,tn)
[*,*]:p(t1,…,tn)
[*,*,*]:p(t1,…,tn)
…
…
```

and

```
[#]:p(t1,…,tn)
[#,#]:p(t1,…,tn)
[#,#,#]:p(t1,…,tn)
…
…
```

In each of these complex terms, the number of `*`s (or #s) in `p(t1,…,tn)` must be equal the number of `*`s (or #s) before the colon.

Semantics:

Example 1: Consider a simple relational schema:

```
movie(TITLE)
actor(NAME)
acts(TITLE,NAME)
direct\or(TITLE,NAME)
```

and the following predicate:

```
acts([*]:movie(*),A)
```

The complex term appears as the first argument of the predicate and the predicate is to be interpreted as follows:

```
{A | (∀T)(movie(T) → acts(T,A))}
```

i.e. it expresses the set of actor names who act in "all" movies present in the movie table. This set can be evaluated using the relational algebraic expression:

```
acts(T,A) ÷ movie(T)
```

Now consider the following predicate:

```
acts([#]:director(#,'Spielberg'),A)
```

This predicate involves the "#" complex term and is to be interpreted as:

```
{A | (∀T)(acts(T,A) → director(T,'Spielberg'))}
```

i.e. it expresses the set of actor names who act "only" in Spielberg directed movies. This set can be evaluated using the following relational algebraic expression:

```
project[A](acts(T,A)) –
project[A](acts(T,A) join
(select[D<>'Spielberg](directs(T,D))))
```

DatalogQ Queries

Consider the following relational schema:

```
movie(TITLE)
director(TITLE,DIRECTOR)
actor(TITLE,ACTOR)
```

(1) Get actors who act in all movies.

```
answer(A) :- actor([*]:movie(*),A).
```

(2) Get actors who do not act in all movies.

```
answer(A) :- actor(T,A), not actor([*]:movie(*),A).
```

(3) Get directors such that every actor has acted in at least one of his or her movies.

```
aperson(A) :- actor(T,A).
r(A,D) :- actor(T,A), director(T,D).
answer(D) :- r([*]:aperson(*),D).
```

(4) Get pairs of actors who have acted in exactly the same set of movies.

```
answer(A1,A2) :-
  actor([*]:actor(*,A2),A1),
  actor([*]:actor(*,A1),A2),
  A1 < A2.
```