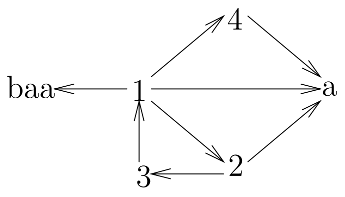
The main task in this assignment is implementing a query engine for non-recursive Datalog, with negation (*nr-Datalog*$^\neg$), as described in the textbook (Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995).

Your query engine should read standard input and write to standard output. The input will define both extensional and intensional database predicates (EDB and IDB predicates) using a rule-based syntax. For example, an EDB predicate `Edge` that represents edges in a directed graph by listing their end points may be defined as indicated on thel left below; the corresponding graph appears on the right for reference.

```
Edge(1,2).  Edge(1,a).
Edge(1,4).  Edge(1,baa).
Edge(2,3).  Edge(2,a).
Edge(3,1).
Edge(4,a).
```



A period (`.` character) is used to denote the end of a statement. Statements may be split over multiple lines, and extra white space between tokens (such as `(`, `1`, `,`, and `2`) is permitted. Thus, the above input may also appear in the following form:

```
Edge (1, 2) .  Edge(1,a) .  Edge(1,
     4).  Edge(1,baa).  Edge( 2,3).
Edge(2,a).  Edge(3,1).  Edge(4,a).
```

We will use the convention that all unadorned strings (such as `1`, `a`, and `baa`) refer to domain elements. Variables all have names that begin with an underscore (_). For example, `_v`, `_answer`, and `__z` refer to variables, but `a_or_b` is a constant. As a special case, a single underscore character denotes an *anonymous variable.* Each occurrence of an underscore character in this context represents a new variable.

IDB predicates are defined using rules based on the above conventions. For example, the following rules define an IDB predicate that lists graph vertices connected by paths of length 2:

```
twoPath(_a, _b) :- Edge(_a, _c), Edge(_c, _b).
twoPath(_a, _b) :- Edge(_b, _c), Edge(_c, _a).
```

The token `:-` is used for the left-arrow symbol (←) in Datalog rules. As another example, the following defines a unary IDB predicate that is satisfied by vertices that have at least one out-edge:

```
hasOut(_v) :- Edge(_v, _).
```

Note the use of an anonymous variable. We will use the convention of naming EDB predicates with an uppercase first letter, while IDB predicates have names beginning with a lowercase letter.

Negated subgoals in rule bodies are indicated by prefixing them with a minus sign. For example, the following defines vertices that have no out-edge (but that have an in-edge):

```
    noOut(_v) :- Edge(_, _v), -Edge(_v, _).
```

Note the use of two anonymous variables; they refer to different variables.

Queries are specified by prefixing an IDB predicate with a question-mark character. For example, the following query should output all vertices in the `noOut` IDB predicate defined above.

```
    ?noOut(_).
```

Queries may contain constants. For example, the following asks for all vertices reachable from the vertex `1` using a path of length 2:

```
    ?twoPath(1, _).
```

Your program should produce one period as output for each period of the input. (For example, each rule, when read by your program, would produce one period as output.) The only other output is that generated by queries. The query result (which is always a relation) should be output one tuple per line, with lines separated by single newline characters. The attributes of a tuple should be separated using single tab characters.

Packaging and Submission: Your submission should contain *only source code and other human-produced files*; it should not contain any object code or other program-generated objects. You should generate a gzip-compressed tar archive file called `M-hw01-N.tgz`, replacing *M* with your last-name and replacing *N* with an arbitrary 4-digit integer (e.g., `Doe-hw01-4242.tgz`). The execution of the following sequence of commands on Gandalf (replacing the file name `Doe-hw01-4242.tgz` with the name of your file) should result in the creation of a directory `/an/arbitrary/dir/Doe-hw01`:

```
    cp Doe-hw01-4242.tgz /an/arbitrary/dir
    cd /an/arbitrary/dir
    gzip -dc Doe-hw01-4242.tgz | tar xf -
```

As the name suggests, `/an/arbitrary/dir` is an *arbitrary* directory that I will select when I test your submission. Typing `make` at the Unix shell prompt in the directory `/an/arbitrary/dir/Doe-hw01` should result in the complete compilation of your program, producing an executable file called `qengine`. You will need to include an appropriate Makefile for this procedure to work. You should also include a short `README` file describing the files in your submission, along with anything that may be helpful in fixing your submission if it does not work as above. You must make sure you program works when stdin and stdout are redirected. For example, we may run your program as follows, where `datafile` is a text file contains the input of the program: `qengine < datafile`. Please check carefully that your file satisfies these requirements.

Submit your file using anonymous FTP (`anonymous` as the user name and your email address as the password) to the FTP server `cs.umaine.edu` in directory `/incoming/cs/atdb/`. If you need to upload an updated version of your submission for any reason, you can follow this procedure again using a different four-digit integer in the file name. (If you try using the same file name as your earlier submission, the upload will likely fail.) You will not be able to list the FTP upload directory (standard security setup), so pay attention to the diagnostic messages from your FTP program. If the messages indicate success, your file will have been uploaded. You must upload the file before you submit your hard-copy homework.