# CHAPTER 2

# Abstract Query Languages

This chapter introduces Java-based interpreters for three abstract query languages: Relational Algebra (RA), Domain Relational Calculus (DRC), and Datalog. The interpreters have been implemented using the parser generator tools JCup and JFlex. In order to use these interpreters, one needs to only download two jar files: `dbengine.jar` and `aql.jar` and include them in the Java CLASSPATH. The JCup libraries are included as part of the jar files and hence the only other software that is required to use the interpreters is a standard Java environment.

The system is simple to use and comes with a database engine that implements a set of basic relational algebraic operators. The interpreter reads a query from the terminal and performs the following three steps:

(1) **Syntax Check**: The query is checked for any syntax errors. If there are any syntactic errors, the interpreter reports these to the terminal and waits to read another query; otherwise the interpreter proceeds to the second step.

(2) **Semantics Check**: The syntactically correct query is checked for semantic errors including type mismatches, invalid column references, and invalid relation names. In addition, the DRC and Datalog interpreters check the queries for safety. If there are any semantic errors or if the DRC/Datalog query is unsafe, the interpreter reports these to the terminal and waits to read another query; otherwise the interpreter proceeds to the third step.

(3) **Query Evaluation**: The query is evaluated using the primitives provided by the database engine and the results are displayed.

## 2.1 Creating the Database

Before the user can start using the interpreters, they must create a database against which they will submit queries. The database consists of several text files all stored within a directory. The directory is named after the database name. For example, to create a database identified with the name `db1` and containing two tables:

```
student(sid:integer,sname:varchar,phone:varchar,gpa:decimal)
skills(sid:integer,language:varchar)
```

a directory called `db1` should be created along with the following three files (one for the catalog description and the remaining two for the data for the two tables):

```
catalog.dat
STUDENT.dat
SKILLS.dat
```

The file names are case sensitive and should strictly follow the convention used, i.e. `catalog.dat` should be all lower case and the data files should be named after their relation name in upper case followed by the file suffix, `.dat`, in lower case.

The `catalog.dat` file contains the number of relations in the first line followed by the descriptions of each relation. The description of each relation begins with the name of the relation in a separate line followed by the number of attributes in a separate line followed by attribute descriptions. Each attribute description includes the name of the attribute in a separate line followed by the data type (`VARCHAR`, `INTEGER`, or `DECIMAL`) in a separate line. All names and data types are in upper case. There should be no leading or trailing white space in any of the lines. The `catalog.dat` file for database `db1` is shown below:

```
2
STUDENT
4
SID
INTEGER
SNAME
VARCHAR
PHONE
VARCHAR
GPA
DECIMAL
SKILLS
2
SID
INTEGER
LANGUAGE
VARCHAR
```

The `db1` directory must include one data file for each relation. In the case of `db1`, they should be named `STUDENT.dat` and `SKILLS.dat`. The data file for relations contains the number of tuples in the first line followed by the description of each tuple. Tuples are described by the values under each column with each vale in a separate line. For example, let the `SKILLS` relation have three tuples:

```
(111,Java)
(111,C++)
(222,Java)
```

These tuples will be represented in the `SKILLS.dat` data file as follows:

```
3
111
Java
```

```
111
C++
222
Java
```

Again, there should be no leading or trailing white spaces in any of the lines. Some pre-defined databases are available along with this laboratory manual. New data may be added to existing databases as well as new databases may be created when needed.

## 2.2 Relational Algebra Interpreter

The RA interpreter is invoked using the following terminal command:

```
$ java edu.gsu.cs.ra.RA 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:

```
RA>
```

At this prompt the user may enter a Relational Algebra query or type the exit command. Every query is terminated by a ";". Even the exit command must end with a semi-colon. Queries may span more than one line; upon typing the ENTER key the interpreter prints the RA> prompt and waits for further input unless the ENTER key is typed after a semi-colon, in which case the query is processed by the interpreter.

### 2.2.1 Relational Algebra Syntax

A subset of Relational Algebra that includes the union, minus, intersect, Cartesian product, natural join, select, project, and rename operators is implemented in the interpreter. The context-free grammar for this subset is shown below:

```
<Query> ::= <Expr> SEMI;
<Expr>  ::= <ProjExpr>  | <RenameExpr>    | <UnionExpr> |
            <MinusExpr> | <IntersectExpr> | <JoinExpr> |
            <TimesExpr> | <SelectExpr>    | RELATION
<ProjExpr>   ::= PROJECT [<AttrList>] (<Expr>)
<RenameExpr> ::= RENAME [<AttrList>] (<Expr>)
<AttrList>   ::= ATTRIBUTE | <AttrList> , ATTRIBUTE
<UnionExpr>  ::= (<Expr> UNION <Expr>)
<MinusExpr>  ::= (<Expr> MINUS <Expr>)
<IntersectExpr>  ::= (<Expr> INTERSECT <Expr>)
```

```
<JoinExpr>   ::= (<Expr> JOIN <Expr>)
<TimesExpr>  ::= (<Expr> TIMES <Expr>)
<SelectExpr> ::= SELECT [<Condition>](<Expr>)
<Condition>  ::= <SimpleCondition> |
                 <SimpleCondition> AND <Condition>
<SimpleCondition> ::= <Operand> <Comparison> <Operand>
<Operand>    ::= ATTRIBUTE | STRING-CONST | NUMBER-CONST
<Comparison> ::= < | <= | = | <> | > | >=
```

The terminal strings in the grammar include

- Keywords for the relational algebraic operators: `PROJECT`, `RENAME`, `UNION`, `MINUS`, `INTERSECT`, `JOIN`, `TIMES`, and `SELECT`. These keywords are case-insensitive.
- Logical keyword `AND` (case-insensitive).
- Miscellaneous syntactic character strings such as (, ), <, <=, =, <>, >, >=, ;, and comma (,).
- Name strings: `RELATION` and `ATTRIBUTE` (case-insensitive names of relations and their attributes).
- Constant strings: `STRING-CONST` (a string enclosed within single quotes; e.g. 'Thomas') and `NUMBER-CONST` (integer as well as decimal numbers; e.g. 232 and -36.1).

An example of a well-formed syntactically correct query for the `company` database of the Elmasri/Navathe text is:

```
( project[ssn](select[lname='Jones'](employee))
   union
  project[superssn](select[dno=5](employee))
);
```

All relational algebra queries must be terminated by a ";".

A relational algebra query in the simplest form is a "relation name". For example the following terminal session with the interpreter illustrates the execution of this simple query form:

```
$ java edu.gsu.cs.ra.RA company
RA> departments;
SEMANTIC ERROR in RA Query: Relation DEPARTMENTS does not exist
RA> department;
DEPARTMENT(DNAME:VARCHAR,DNUMBER:INTEGER,MGRSSN:VARCHAR,MGRSTARTDA
TE:VARCHAR)

Number of tuples = 6
Research:5:333445555:22-MAY-1978:
Administration:4:987654321:01-JAN-1985:
Headquarters:1:888665555:19-JUN-1971:
Software:6:111111100:15-MAY-1999:
Hardware:7:444444400:15-MAY-1998:
```

```
Sales:8:555555500:01-JAN-1997:

RA> exit;
$
```

In response to a query, the interpreter displays the schema of the result followed by the answer to the query. Individual values within a tuple are terminated by a ":". The simplest query form is useful to display the database contents.

More complicated relational algebra queries involve one or more applications of one or more of the several operators such as select, project, times, join, union, etc. For example, consider the query "*Retrieve the names of all employees working for Dept. No. 5*". This would be expressed by the query execution in the following RA session:

```
RA> project[fname,lname](select[dno=5](employee));
temp1(FNAME:VARCHAR,LNAME:VARCHAR)

Number of tuples = 4
Franklin:Wong:
John:Smith:
Ramesh:Narayan:
Joyce:English:

RA>
```

### 2.2.2 Naming of Intermediate Relations and Attributes

The RA interpreter assigns temporary relation names such as `temp0, temp1`, etc. to each intermediate relation encountered in the execution of the entire query. The RA interpreter also employs the following rules as far as naming of attributes/columns of intermediate relations:

1. Union, Minus, and Intersect: The attribute/column names from the left operand are used to name the attributes of the output relation.
2. Times (Cartesian Product): Attribute/Column names from both operands are used to name the attributes of the output relation. Attribute/Column names that are common to both operands are prefixed by relation name (`tempN`).
3. Select: The attribute names of the output relation are the same as the attribute/column names of the operand.
4. Project, Rename: Attribute/Column names present in the attribute list parameter of the operator are used to name the attributes of the output relation. Duplicate attribute/column names are not allowed in the attribute list.
5. Join (Natural Join): Attribute/Column names from both operands are used to name the attributes of the output relation. Common attribute/column names appear only once.

As another example, consider the query "*Retrieve the social security numbers of employees who either work in department 5 or directly supervise an employee who works in department 5*". The query is illustrated in the following RA session:

```
RA> (project[ssn](select[dno=5](employee))
RA> union project[superssn](select[dno=5](employee)));
temp4(SSN:VARCHAR)

Number of tuples = 5
333445555:
123456789:
666884444:
453453453:
888665555:

RA>
```

### 2.2.3 Relational Algebraic Operators Supported by the RA Interpreter

**Select:** As can be noted from the grammar, the select operator supported by the interpreter has the following syntax:

```
select[condition](expression)
```

where `condition` is a conjunction of one or more simple conditions involving comparisons of attributes or constants with other attributes or constants. The attributes used in the condition must be present in the attributes of the relation corresponding to `expression`.

**Project**: The project operator supported by the interpreter has the following syntax:

```
project[attribute-list](expression)
```

where `attribute-list` is a comma separated list of attributes, each of which is present in the attributes of the relation corresponding to `expression`.

**Rename**: The syntax for the rename operator is

```
rename[attribute-list](expression)
```

where `attribute-list` is a comma separated list of attribute names. The number of attributes mentioned in the list must be equal to the number of attributes of the relation corresponding to `expression`.

**Join**: The syntax for the join operator is

```
(expression1 join expression2)
```

There is no restriction on the schemas of the two expressions.

**Times**: The syntax for the times operator is

```
(expression1 times expression2)
```

There is no restriction on the schemas of the two expressions.

**Union**: The syntax for the union operator is

```
(expression1 union expression2)
```

The schemas of the two expressions must be compatible (same number of attributes and same data types; the names of the attributes may be different).

**Minus**: The syntax for the minus operator is

```
(expression1 minus expression2)
```

The schemas of the two expressions must be compatible (same number of attributes and same data types; the names of the attributes may be different).

**Intersect**: The syntax for the intersect operator is

```
(expression1 intersect expression2)
```

The schemas of the two expressions must be compatible (same number of attributes and same data types; the names of the attributes may be different).

### 2.2.4 Examples

The queries from Section 6.5 of the Elmasri/Navathe text modified to work with the RA interpreter are shown below:

**Query 1**: Retrieve the name and address of employees who work for the "`Research`" department.

```
project[fname,lname,address](
  (rename[dname,dno,mgrssn,mgrstartdate](
      select[dname='Research'](department))
   join
   employee
  )
);
```

**Query 2**: For every project located in "Stafford", list the project number, the controlling department number, and the department manager's last name, address, and birth date.

```
project[pnumber,dnum,lname,address,bdate](
  (
   (select[plocation='Stafford'](projects)
    join
    rename[dname,dnum,ssn,mgrstartdate](department)
   )
   join employee
  )
);
```

**Query 3**: Find the names of employees who work on all the projects controlled by department number 5.

```
project[lname,fname](
 (employee
  join
  (project[ssn](employee)
   minus
   project[ssn](
     (
       (project[ssn](employee)
        times
        project[pnumber](select[dnum=5](projects))
       )
       minus
       rename[ssn,pnumber](project[essn,pno](works_on))
     )
   )
  )
 )
);
```

**Query 4**: Make a list of project numbers for projects that involve an employee whose last name is "Smith", either as a worker or as a manager of the department that controls the project.

```
( project[pno](
   (rename[essn](project[ssn](select[lname='Smith'](employee)))
    join
    works_on
   )
  )
 union
  project[pnumber](
   ( rename[dnum](project[dnumber](select[lname='Smith'](
```

```
        (employee
         join
         rename[dname,dnumber,ssn,mgrstartdate](department)
        )
        )
        )
      )
      join
      projects
     )
  )
);
```

**Query 5**: List the names of all employees with two or more dependents.

```
project[lname,fname](
(rename[ssn](
 project[essn1](
  select[essn1=essn2 and dname1<>dname2](
   (rename[essn1,dname1](project[essn,dependent_name](dependent))
    times
    rename[essn2,dname2](project[essn,dependent_name](dependent)))
   )
  )
 )
join
employee)
);
```

**Query 6**: Retrieve the names of employees who have no dependents.

```
project[lname,fname](
 ( ( project[ssn](employee)
     minus project[essn](dependent)
   )
   join
   employee
 )
);
```

**Query 7**: List the names of managers who have at least one dependent.

```
project[lname,fname](
  ((rename[ssn](project[mgrssn](department))
    join
    rename[ssn](project[essn](dependent))
   )
```

```
  join
  employee
  )
);
```

**Important Tip**: Since many of the queries shown above are long and span multiple lines, the best way to use the interpreter is to create a text file in which the queries are typed. These queries are then cut and pasted into the interpreter prompt. Any errors in syntax or semantics should be corrected in the text file and then the process of cut and paste should be repeated until a correct solution is reached.

## 2.3 Domain Relational Calculus Interpreter

The DRC interpreter is invoked using the following terminal command:

```
$ java edu.gsu.cs.drc.DRC 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:

```
DRC>
```

At this prompt the user may enter a Domain Relational Calculus query or type the `exit` command. Each DRC query is expressed in a set-notation using a pair of curly brackets as follows:

```
{ variable-list | P(variable-list) }
```

where `variable-list` is a comma-separated list of variables which must all be present in the body predicate of the query `P(variable-list)` as free-variables.

The `exit` command must end with a semi-colon. Queries may span more than one line; upon typing the `ENTER` key the interpreter prints the `DRC>` prompt and waits for further input unless the `ENTER` key is typed after a right curly bracket (`}`), in which case the query is processed by the interpreter.

### 2.3.1 Domain Relational Calculus Syntax

The context-free grammar for DRC queries implemented within the DRC interpreter is shown below:

```
Query ::= LBRACE VarList BAR Formula RBRACE;
```

```
VarList ::= NAME | VarList COMMA NAME;
Formula ::= AtomicFormula |
            Formula AND Formula |
            Formula OR Formula |
            NOT LPAREN Formula RPAREN |
            LPAREN EXISTS VarList RPAREN LPAREN Formula RPAREN |
            LPAREN FORALL VarList RPAREN LPAREN Formula RPAREN;
AtomicFormula ::=
     NAME LPAREN ArgList RPAREN | Arg Comparison Arg;
ArgList ::= Arg | ArgList COMMA Arg;
Arg ::= NAME | STRING | NUMBER;
Comparison ::= < | <= | = | <> | > | >=
```

The terminal strings in the grammar include

- Keywords for the logical operators: `AND`, `OR`, and `NOT`. These keywords are case-insensitive.
- Quantifier keywords `EXISTS` and `FORALL` (case-insensitive).
- Miscellaneous syntactic character strings such as (, ), <, <=, =, <>, >, >=, and comma (,).
- `NAME` strings: used for named relations and variables (case-insensitive).
- Constant strings: `STRING` (a string enclosed within single quotes; e.g. 'Thomas') and `NUMBER` (integer as well as decimal numbers; e.g. 232 and -36.1).

An example of a well-formed syntactically correct query on the `company` database of the Elmasri/Navathe text is:

```
{ x | (exists a1,a2,a3,a4,a5,a6,a7,a8)(
          employee(a1,a2,'Jones',x,a3,a4,a5,a6,a7,a8)) or
      (exists a1,a2,a3,a4,a5,a6,a7,a8)(
          employee(a1,a2,a3,x,a4,a5,a6,a7,a8,5)) }
```

All DRC queries must be enclosed within a pair of matching curly brackets.

The simplest DRC query displays the contents of a relation. For example the following terminal session with the interpreter illustrates the execution of this simple query form that displays the contents of the `DEPARTMENT` relation:

```
$ java edu.gsu.cs.drc.DRC company
DRC> { a,b,c,d | department(a,b,c,d) }
ANSWER(A:VARCHAR,B:INTEGER,C:VARCHAR,D:VARCHAR)

Number of tuples = 6
Research:5:333445555:22-MAY-1978:
Administration:4:987654321:01-JAN-1985:
Headquarters:1:888665555:19-JUN-1971:
Software:6:111111100:15-MAY-1999:
```

```
Hardware:7:444444400:15-MAY-1998:
Sales:8:555555500:01-JAN-1997:

DRC>
```

In response to a query, the interpreter displays the schema of the result followed by the answer to the query.

### 2.3.2 Safe DRC Queries

The DRC interpreter checks for the "safety" of queries and evaluates only those that are determined to be safe. An error message is generated for unsafe queries.

For the discussion of safe DRC queries, we will assume that the formula defining the query does not contain the `forall` quantifier. If the `forall` quantifier does appear in the formula, the user can convert such a formula to an equivalent one without the `forall` quantifier using the logical equivalence:

```
(forall X)(F) ≡ NOT ((exists X)( NOT (F)))
```

It is almost always the case that the `F` in the `forall` quantified formula above is of the form

```
NOT (P) or Q
```

In case the user does not eliminate the `forall` quantifier, the DRC interpreter would automatically convert all `forall` quantified formulas into equivalent `exists` quantified formulas using the above equivalence. In addition, the interpreter would also apply the DeMorgan's law:

```
NOT (P or Q) ≡ NOT (P) and NOT (Q)
```

to push the `NOT` further inside the formula.

As an example of this automatic transformation, consider the following query provided by the user:

```
{a,b |(exists c)(r(a,b,c) and
          (forall d,e)(not(s(a,d,e)) or (exists f)(t(d,f)))) }
```

The DRC interpreter would convert the above query to:

```
{a,b |(exists c)(r(a,b,c) and
        not(exists d,e)(s(a,d,e) and not(exists f)(t(d,f)))) }
```

**Definition:** A DRC query (without `forall` quantifiers) is defined to be *safe* if it satisfies the following three conditions:

(a) For every sub-formula in the query connected with an "`or`", the two operand formulas have the same set of free variables, i.e. the "`or`" formula is of the form:

```
F(X1,…,Xn) or G(X1,…,Xn)
```

(b) All free variables appearing in "maximal sub-conjuncts", `F1 and … and Fn`, must be "limited" in that they either appear in (i) a positive sub-formula `Fi` or (ii) as `X` in an sub-formula of the form `X=a` or `a=X` or (iii) as `X` in a sub-formula of the form `X=Y` where `Y` is determined to be "limited".

(c) The NOT operator may be applied only to a term in a maximal sub-conjunct of type discussed in (b), i.e. all free variables in the NOT term must be shown to be "limited" in the positive terms of the maximal sub-conjunct.

Some examples follow. The following query would be considered safe as it satisfies condition (a).

```
{a,b | (exists c)(r(a,b,c)) or s(a,b) }
```

But the following would not be safe:

```
{a,b | (exists b,c)(r(a,b,c)) or s(a,b) }
```

This is because the free variables on the left operand of the "`or`" formula consists of only one variable, `a`, and the free variables on the right operand consists of two variables, `a` and `b`.

The query formula from an earlier query:

```
(exists c)(r(a,b,c) and
  not(exists d,e)(s(a,d,e) and not(exists f)(t(d,f))))
```

is safe. The formula has the following two maximal sub-conjuncts (ignoring atomic formulas which are maximal sub-conjuncts of size 1):

(1) `s(a,d,e) and not(exists f)(t(d,f))`
all three free variables `a`, `d`, and `e` are limited as they appear in `s(a,d,e)`

(2) `r(a,b,c) and`
`not(exists d,e)(s(a,d,e) and not(exists f)(t(d,f)))`
all three free variables `a, b,` and `c` are limited as they appear in `r(a,b,c).`

The free variables in each of the maximal sub-conjuncts are shown to be "limited" and hence the overall query is safe.

The following query formula is unsafe:

```
p(a,b) and not ((exists c)(q(b,c,d)))
```

This is because the free variable `d` is not “limited” as it is not grounded in a positive term in the maximal sub-conjunct.

### 2.3.3 DRC Query Examples

The queries from Section 6.7 of the Elmasri/Navathe text modified to work with DRC interpreter are shown below:

**Query 0:** Retrieve the birthdate and address of the employees whose name is "`John B. Smith`".

```
{ u,v | (exists t,w,x,y,z)(
            employee('John','B','Smith',t,u,v,w,x,y,z) ) }
```

**Query 1**: Retrieve the name and address of all employees who work for the "`Research`" department.

```
{ q,s,v | (exists r,t,u,w,x,y,z,n,o)(
            employee(q,r,s,t,u,v,w,x,y,z) and
            department('Research',z,n,o) ) }
```

**Query 2**: For every project located in "`Stafford`", list the project number, the controlling department number, and the department manager's last name, birth date, and address.

```
{ i,k,s,u,v | (exists h,q,r,t,w,x,y,z,l,o)(
                projects(h,i,'Stafford',k) and
                employee(q,r,s,t,u,v,w,x,y,z) and
                department(l,k,t,o) ) }
```

**Query 6**: List the names of employees who have no dependents.

```
{ q,s | (exists r,t,u,v,w,x,y,z)(
          employee(q,r,s,t,u,v,w,x,y,z) and
          not ((exists m,n,o,p)(dependent(t,m,n,o,p))) ) }
```

The following is not SAFE and would not work

```
{ q,s | (exists r,t,u,v,w,x,y,z)(
          employee(q,r,s,t,u,v,w,x,y,z) and
         (forall l,m,n,o,p)(not (dependent(l,m,n,o,p)) or t<>l) )}
```

**Query 7**: List the names of managers who have at least one dependent.

```
{ s,q | (exists r,t,u,v,w,x,y,z,h,i,k,m,n,o,p)(
           employee(q,r,s,t,u,v,w,x,y,z) and
           department(h,i,t,k) and
           dependent(t,m,n,o,p) ) }
```

## 2.4 Datalog Interpreter

The DLOG interpreter is invoked using the following terminal command:

$ `java edu.gsu.cs.dlg.DLOG 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:

```
DLOG>
```

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

### 2.4.1 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.

### 2.4.2 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 edu.gsu.cs.dlg.DLOG 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...

[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q2;
-----------------------------------
answer(F,M,L) :-
  employee(F,M,L,S,_,_,_,_,_,_),
  dependent(S,F,_,_,_).$
-----------------------------------
ANSWER(F:VARCHAR,M:VARCHAR,L:VARCHAR)

Number of tuples = 1
Alec:C:Best:

DLOG> exit;
Exiting...
```

```
[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q3;
-----------------------------------
answer(F,M,L) :-
  employee(F,M,L,_,_,_,_,_,S,_),
  employee('Franklin',_,'Wong',S,_,_,_,_,_,_).$
-----------------------------------
ANSWER(F:VARCHAR,M:VARCHAR,L:VARCHAR)

Number of tuples = 3
John:B:Smith:
Ramesh:K:Narayan:
Joyce:A:English:

DLOG> exit;
Exiting...

[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q4;
-----------------------------------
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).$
-----------------------------------
ANSWER(F:VARCHAR,M:VARCHAR,L:VARCHAR)

Number of tuples = 0

DLOG> exit;
Exiting...

[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q5;
-----------------------------------
```

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

Number of tuples = 2
Bob:B:Bender:
Kate:W:King:

DLOG> exit;
Exiting...

[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q6;
-----------------------------------
temp1(S) :-
  works_on(S,P,_), projects(_,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).$
-----------------------------------
ANSWER(F:VARCHAR,M:VARCHAR,L:VARCHAR,A:VARCHAR)

Number of tuples = 1
Jennifer:S:Wallace:291 Berry, Bellaire, TX:

DLOG> exit;
Exiting...
[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q7;
-----------------------------------
temp1(S) :-
  works_on(S,P,_),
  projects(_,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).$
-----------------------------------
ANSWER(F:VARCHAR,M:VARCHAR,L:VARCHAR,A:VARCHAR)

Number of tuples = 38
James:E:Borg:450 Stone, Houston, TX:
Franklin:T:Wong:638 Voss, Houston, TX:
Jennifer:S:Wallace:291 Berry, Bellaire, TX:
Ramesh:K:Narayan:971 Fire Oak, Humble, TX:
Alicia:J:Zelaya:3321 Castle, Spring, TX:
Ahmad:V:Jabbar:980 Dallas, Houston, TX:
Jared:D:James:123 Peachtree, Atlanta, GA:
Alex:D:Freed:4333 Pillsbury, Milwaukee, WI:
John:C:James:7676 Bloomington, Sacramento, CA:
Jon:C:Jones:111 Allgood, Atlanta, GA:
Justin:null:Mark:2342 May, Atlanta, GA:
Brad:C:Knight:176 Main St., Atlanta, GA:
Evan:E:Wallis:134 Pelham, Milwaukee, WI:
Josh:U:Zell:266 McGrady, Milwaukee, WI:
Andy:C:Vile:1967 Jordan, Milwaukee, WI:
Tom:G:Brand:112 Third St, Milwaukee, WI:
Jenny:F:Vos:263 Mayberry, Milwaukee, WI:
Chris:A:Carter:565 Jordan, Milwaukee, WI:
Kim:C:Grace:6677 Mills Ave, Sacramento, CA:
Jeff:H:Chase:145 Bradbury, Sacramento, CA:
Bonnie:S:Bays:111 Hollow, Milwaukee, WI:
Alec:C:Best:233 Solid, Milwaukee, WI:
Sam:S:Snedden:987 Windy St, Milwaukee, WI:
Nandita:K:Ball:222 Howard, Sacramento, CA:
Bob:B:Bender:8794 Garfield, Chicago, IL:
Jill:J:Jarvis:6234 Lincoln, Chicago, IL:
Kate:W:King:1976 Boone Trace, Chicago, IL:
Lyle:G:Leslie:417 Hancock Ave, Chicago, IL:
Billie:J:King:556 Washington, Chicago, IL:
Jon:A:Kramer:1988 Windy Creek, Seattle, WA:
Ray:H:King:213 Delk Road, Seattle, WA:
Gerald:D:Small:122 Ball Street, Dallas, TX:
Arnold:A:Head:233 Spring St, Dallas, TX:
Helga:C:Pataki:101 Holyoke St, Dallas, TX:
Naveen:B:Drew:198 Elm St, Philadelphia, PA:
Carl:E:Reedy:213 Ball St, Philadelphia, PA:
Sammy:G:Hall:433 Main Street, Miami, FL:
Red:A:Bacher:196 Elm Street, Miami, FL:

DLOG> exit;
Exiting...
```

```
[raj@tinman ch2]$ java edu.gsu.cs.dlg.DLOG company
type "help;" for usage...
Message: Database Provided: Database Directory is ./company
DLOG> @q8;
-----------------------------------
temp1(S) :-
  dependent(S,_,_,_,_).
answer(L) :-
  employee(_,_,L,S,_,_,_,_,_,_),
  department(_,_,S,_),
  not temp1(S).$
-----------------------------------
ANSWER(L:VARCHAR)

Number of tuples = 2
Borg:
James:

DLOG> exit;
Exiting...
[raj@tinman ch2]$
```

## Exercises

1. Specify and execute the following queries using the RA interpreter on the COMPANY database schema.

    a. Retrieve the names of all employees in department 5 who work more than 10 hours per week on the 'ProductX' project.
    b. List the names of all employees who have a dependent with the same first name as themselves.
    c. Find the names of employees who are directly supervised by 'Franklin Wong'.
    d. Retrieve the names of employees who work on every project.
    e. Retrieve the names of employees who do not work on any project.
    f. Retrieve the names and addresses of all employees who work on at least one project located in Houston but whose department has no location in Houston.
    g. Retrieve the last names of all department managers who have no dependents.

2. Consider the following MAILORDER relational schema describing the data for a mail order company:

```
parts(pno,pname,qoh,price,olevel)
customers(cno,cname,street,zip,phone)
employees(eno,ename,zip,hdate)
zipcodes(zip,city)
```

```
orders(ono,cno,eno,received,shipped)
odetails(ono,pno,qty)
```

The attribute names are self-explanatory. "qoh" stands for quantity on hand. Specify and execute the following queries using the RA interpreter on the MAILORDER database schema.

a. Retrieve the names of parts that cost less than $20.00.
b. Retrieve the names and cities of employees who have taken orders for parts costing more than $50.00
c. Retrieve the pairs of customer number values of customers who live in the same zip code.
d. Retrieve the names of customers who have ordered parts only from employees living in the city of Wichita.
e. Retrieve the names of customers who have ordered all parts costing less than $20.00.
f. Retrieve the names of customers who have not placed a single order.
g. Retrieve the names of customers who have placed exactly two orders.

3. Consider the following GRADEBOOK relational schema describing the data for a grade book of a particular instructor (Note: The attributes `A`, `B`, `C`, and `D` store grade cutoffs.)

```
catalog(cno,ctitle)
students(sid,fname,lname,minit)
courses(term,secno,cno,A,B,C,D)
enrolls(sid,term,secno)
```

Specify and execute the following queries using the RA interpreter on the GRADEBOOK database schema.

a. Retrieve the names of students enrolled in the '`Automata`' class in the term of `Fall 1996`.
b. Retrieve the SID values of students who have enrolled in `CSc226` as well as `CSc227`.
c. Retrieve the SID values of students who have enrolled in `CSc226` or `CSc227`.
d. Retrieve the names of students who have not enrolled in any class.
e. Retrieve the names of students who have enrolled in all courses in the `catalog` table.

4. Consider the database consisting of the following relations:

```
supplier(sno, sname)
part(pno, pname)
project(jno, jname)
supply(sno,pno,jno)
```

The database records information about suppliers, parts, and projects and includes a ternary relationship between suppliers, parts, and projects. This relationship is a many-many-many relationship. Specify and execute the following queries using the RA interpreter.

a. Retrieve part numbers of parts that are supplied to exactly two projects.
b. Retrieve supplier names of suppliers who supply more than two parts to project '*J1*'.
c. Retrieve part numbers of parts that are supplied by every supplier.
d. Retrieve project names of projects that are supplied only by suppliers '*S1*'.
e. Retrieve supplier names of suppliers who supply at least two different parts each to at least two different projects.

5. Specify and execute the following queries for the database in Exercise 5.16 of the Elmasri/Navathe text using the RA interpreter.

    a. Retrieve the names of students who have enrolled in a course that uses a textbook published by Addison Wesley.
    b. Retrieve the course names of courses which have changed their textbook at least once.
    c. Retrieve the names of departments that only adopt textbooks from Addison Wesley.
    d. Retrieve the names of departments that have adopted all textbooks written by Navathe and published by Addison Wesley in their courses.
    e. Retrieve the names of students who have never used a book (in a course) written by Navathe and published by Addison Wesley.

6. Repeat Exercises 1 through 5 in domain relational calculus (DRC) by using the DRC interpreter.

7. Repeat Exercises 1 through 5 in Datalog (DLOG) by using the DLOG interpreter.