1. Write the query of Exercise 6.19 using TRC and DRC: Find the names of all brokers who have made money in all accounts assigned to them.

   TRC: $\{\texttt{B.Name} \mid \textsc{Broker}(\texttt{B})\ \text{AND}$
   $\quad \forall \texttt{A} \in \textsc{Account}\,(\texttt{A.BrokerId} = \texttt{B.Id} \ \rightarrow \texttt{A.Gain} > 0)$

   DRC: $\{\texttt{Name} \mid \exists \texttt{BrokerId}\,(\textsc{Broker}(\texttt{BrokerId}, \texttt{Name})\ \text{AND}$
   $\quad \forall \texttt{Acc\#}\,\forall \texttt{Gain}\,(\textsc{Account}(\texttt{Acc\#}, \texttt{BrokerId}, \texttt{Gain}) \ \rightarrow\ \texttt{Gain} > 0\,)\,)\ \}$

2. Express the following query in TRC and DRC where the schema is given in Figure 4.4, page 62: Find all courses in department `MGT` that were taken by all students.

   TRC:

   $\{\texttt{C.CrsCode}, \texttt{C.CrsName} \mid \textsc{Course}(\texttt{C})\ \text{AND}\ \texttt{C.DeptId = 'MGT'}\ \text{AND}$
   $\quad \forall S \in \textsc{Student}\ \exists R \in \textsc{Transcript}$
   $\quad (\texttt{R.StudId} = \texttt{S.Id}\ \text{AND}\ \texttt{R.CrsCode} = \texttt{C.CrsCode})\ \}$

   DRC:

   $\{\texttt{CrsCode}, \texttt{CrsName} \mid \exists \texttt{Descr}\ (\textsc{Course}(\texttt{'MGT',CrsCode,CrsName,Descr}))\ \text{AND}$
   $\quad \forall \texttt{Id} \in \textsc{Student}.\texttt{Id}\ \exists \texttt{Semester} \exists \texttt{Grade}$
   $\quad\quad (\textsc{Transcript}(\texttt{Id,CrsCode,Semester,Grade})\ )\ \}$

3. Consider a relation DIRECTFLIGHT(`StartCity, DestinationCity`) that lists all direct flights among cities. Use the recursion facility of SQL:1999 to write a query that finds all pairs $\langle city_1, city_2 \rangle$ such that there is an *indirect* flight from $city_1$ to $city_2$ with at least two stops in-between.

   The simplest way to do this is to compute INDIRECTFLIGHT similarly to INDIRECTPREREQVIEW:

   ```
   CREATE RECURSIVE VIEW IndirectFlight(From, To) AS
           SELECT * FROM DirectFlight
           UNION
           SELECT P.StartCity, I.To
           FROM DirectFlight D, IndirectFlight I
           WHERE D.DestinationCity = I.From
   ```

   Then we can compute all flights with just one stop — FLIGHT1 — and then subtract DIRECTFLIGHT and FLIGHT1 from INDIRECTFLIGHT.

   One might be tempted to first create a recursive view INDIRECTFLIGHT2(`From, TO, NumberOfStops`) and then select the flights with `NumberOfStops > 1`.

   ```
   CREATE RECURSIVE VIEW IndirectFlight2(From, To, Stops) AS
           SELECT D.StartCity, D.DestinationCity, 0
           FROM DirectFlight D
           UNION
           SELECT P.StartCity, I.To, I.Stops+1
           FROM DirectFlight D, IndirectFlight2 I
           WHERE D.DestinationCity = I.From
   ```

However, this recursive definition has a problem: because we keep incrementing the number of stops with each iteration, the evaluation process will not terminate. (Check that the termination condition will never be true!)

To avoid the problem, one would have to check, in a nested subquery, that each newly generated tuple, $\langle to, from, ?\rangle$ has not been seen before for some number of stops. Unfortunately this would introduce a non-stratified use of negation. To avoid this problem, we can compute IndirectFlight (the 2-column version) first and then check the $\langle to, from\rangle$ pairs against that view.

It is easy to see that the first solution is faster, if we only need to find flights that have at least 2 stops and nothing else. However, if we expect queries about the flights with 1 stop, 2 stops, etc., then the second approach is more attractive.

4. Consider an Account class and a TransactionActivity class in a banking system.

   (a) Posit ODMG ODL class definitions for them. The Account class must include a relationship to the set of objects in the TransactionActivity class corresponding to the deposit and withdraw transactions executed against that account.

```
class  Account {
    attribute  Integer AcctId;
    relationship  Set< Person> Owner;
    relationship  Set< TransactionActivity> Transactions
                inverse  TransactionActivity::ActivityAccount;
}

class  TransactionActivity {
    attribute  Integer  Type;  // assume type is some integer code
    attribute  Float   Amount;
    attribute  Date   ActivityDate;
    relationship  Account ActivityAccount
                    inverse  Account::Transactions;
}
```

   (b) Give an example of an object instance satisfying that description.

```
(#123,   [12345,
           {(#p4345, [123456789, "John Doe"]),
              (#p0987, [654345643, "Mary Doe"])},
           {(#t435, [2, 58.34, 2001-3-4, #123]),
              (#t8132, [1, 231.99, 2001-4-5, #123])}
         ])
....
```

   (c) Give an example of an OQL query against that database, which will return the account numbers of all accounts for which there was at least one withdrawal of more than $10,000.

```
SELECT A.AcctId
FROM   AccountExt A
         // assume the withdrawal code is 2
```

```
WHERE  flatten(A.Transactions).Type = 2
     AND  flatten(A.Transactions).Amount > 10000
```

5. Write an OQL query that, for each major, computes the number of students who have that major. Use the STUDENT class defined in (16.8) on page 502.

```
SELECT Major: S.Major,  count: count(S2.Id)
FROM  StudentExt S,  StudentExt S2
WHERE  S.Major = S2.Major
GROUP BY S.Major
```

We could also write this query using nested queries as follows:

```
SELECT DISTINCT Major: S.Major,
            count:  count( SELECT S2.Id
                           FROM  StudentExt S2
                           WHERE S2.Major = S.Major)

FROM  StudentExt S
```

6. Use the schema defined for the exercise 16.18 to answer the following queries:

   (a) Find all students who have taken more than five classes in the mathematics department.

```
SELECT S.Name
FROM  Student S
WHERE  5 < ( SELECT count(S1.Transcript->Course)
             FROM    Student S1
             WHERE  S1.Transcript->Course.DeptId = 'MAT'
                    AND  S1.Id = S.Id)
```

   (b) Represent grades as a UDT, GRADE, with a method, `value()`, that returns the grade's numeric value.

```
CREATE TYPE  GradeType  AS (
   LetterValue  CHAR(2) )
METHOD value()  RETURNS DECIMAL(3);

CREATE METHOD value()  FOR  GradeType
RETURNS DECIMAL(3)
LANGUAGE SQL
   BEGIN
      IF LetterValue = 'A'  THEN RETURN 4;
      IF LetterValue = 'A-'  THEN RETURN 3.66;
      ... ... ...
   END
```

   (c) Write a method that, for each student, computes the average grade. This method requires the `value()` method that you constructed for the previous problem.

```
SELECT S.Name, avg(S.Transcript.Grade.LetterValue.value())
FROM   STUDENT S
```

7. Specify a DTD appropriate for a document that contains data from both the COURSE table in Figure 5.15, page 116, and the REQUIRES table in Figure 5.16, page 117. Try to reflect as many constraints as the DTDs allow. Give an example of a document that conforms to your DTD.

   One good example of such a document is shown below. Note that it does not try to mimic the relational representation, but instead courses are modeled using the object-oriented approach.

```
<Courses>
   <Course CsrCode="CS315" DeptId="CS"
           CrsName="Transaction Processing" CreditHours="3">
      <Prerequisite CrsCode="CS305" EnforcedSince="2000/08/01"/>
      <Prerequisite CrsCode="CS219" EnforcedSince="2001/01/01"/>
   </Course>
   <Course>
      .....
   </Course>
   .....
</Courses>
```

   An appropriate DTD would be:

```
<!DOCTYPE Courses [
   <!ELEMENT Courses (Course*)>
   <!ELEMENT Course (Prerequisite*)>
   <!ATTLIST Course
             CrsCode ID #REQUIRED
             DeptId  CDATA #IMPLIED
             CrsName CDATA #REQUIRED
             CreditHours CDATA #REQUIRED >
   <!ATTLIST Prerequisite
             CrsCode IDREF #REQUIRED
             EnforcedSince CDATA #REQUIRED>
]>
```

8. Define the following simple types:

   (a) A type whose domain consists of lists of strings, where each list consists of 7 elements.

```
<simpleType name="ListsOfStrings">
   <list itemType="string" />
</simpleType>
<simpleType name="ListsOfLength7">
   <restriction base="ListsOfStrings">
        <length value="7"/>
   </restriction>
</simpleType>
```

(b) A type whose domain consists of lists of strings, where each string is of length 7.

```
<simpleType name="StringsOfLength7">
   <restriction base="string">
      <length value="7"/>
   </restriction>
</simpleType>
<simpleType name="ListsOfStringsOfLength7">
   <list itemType="StringsOfLength7" />
</simpleType>
```

(c) A type appropriate for the letter grades that students receive on completion of a course—A, A−, B+, B, B−, C+, C, C−, D, and F. Express this type in two different ways: as an enumeration and using the `pattern` tag of XML Schema.

```
<simpleType name="gradesAsEnum">
   <restriction base="string">
      <enumeration value="A"/>
      <enumeration value="A-"/>
      <enumeration value="B+"/>
      ..........
   </restriction>
</simpleType>
<simpleType name="gradesAsPattern">
   <restriction base="string">
      <pattern value="(A-?|[BC][+-]?|[DF])"/>
   </restriction>
</simpleType>
```

In the `gradesAsPattern` representation, (...) represent complex alternatives of patterns separated by | (W3C has adopted the syntax of regular expressions used in Perl), [...] represent simple alternatives (of characters), and ? represents zero or one occurrence, as usual in regular expressions.

9. Write an XML schema specification for a simple document that lists stock brokers with the accounts that they handle and a separate list of the client accounts. The Information about the accounts includes the account `Id`, ownership information, and the account positions (*i.e.*, stocks held in that account). To simplify the matters, it suffices, for each account position, to list the stock symbol and quantity. Use `ID`, `IDREF`, and `IDREFS` to specify referential integrity.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:brk="http://somebrokerage.com/documents"
        targetNamespace="http://somebrokerage.com/documents">
  <element name="Brokerage">
    <complexType>
      <sequence>
        <element name="Broker" type="brk:brokerType"
                 minOccurs="0" maxOccurs="unbounded"/>
        <element name="Account" type="brk:accountType"
                 minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>
  <complexType name="brokerType">
    <attribute name="Id" type="ID" />
    <attribute name="Name" type="string" />
    <attribute name="Accounts" type="IDREFS" />
  </complexType>
  <complexType name="accountType">
    <attribute name="Id" type="ID" />
    <attribute name="Owner" type="string" />
    <element name="Positions" />
      <complexType>
        <sequence>
          <element name="Position">
            <complexType>
              <attribute name="stockSym" type="string" />
              <attribute name="qty" type="integer" />
            <complexType>
          </element>
        </sequence>
      </complexType>
    </element>
  </complexType>
</schema>
```