DTB

```
<!ELEMENT Course_Catalog (Department*)>
<!ELEMENT Department (Title,Chair,Course*)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Chair (Professor)>
<!ELEMENT Professor (First_Name,Middle_Initial?, Last_Name)>
<!ELEMENT First_Name (#PCDATA)>
<!ELEMENT Middle_Initial (#PCDATA)>
<!ELEMENT Last_Name (#PCDATA)>
<!ELEMENT Course (Title,Description,Instructors)>
<!ELEMENT Description (#PCDATA)>
<!ELEMENT Instructors (Professor+)>


<!DOCTYPE A [
  <!ELEMENT A (B+, C)>
  <!ELEMENT B (#PCDATA)>
  <!ELEMENT C (B?, D)>
  <!ELEMENT D (#PCDATA)>
]>

<!ATTLIST EMP name IDREF #REQUIRED>
<!ATTLIST PHONE type IDREF #IMPLIED>
<!ATTLIST PHONE owner IDREF #IMPLIED>
<!ATTLIST EMP ssNo CDATA #REQUIRED>

<!DOCTYPE meal [
  <!ELEMENT meal (person*,food*,eats*)>
  <!ELEMENT person EMPTY>
  <!ELEMENT food EMPTY>
  <!ELEMENT eats EMPTY>
  <!ATTLIST person name ID #REQUIRED>
  <!ATTLIST food name ID #REQUIRED>
  <!ATTLIST eats diner IDREF #REQUIRED dish IDREF #REQUIRED>
]>

<meal>
<person name="Alice"/>
<person name="Bob"/>
<person name="Carol"/>
<person name="Dave"/>
<food name="salad"/>
<food name="turkey"/>
<food name="sandwich"/>
<eats diner="Alice" dish="turkey"/>
<eats diner="Bob" dish="salad"/>
<eats diner="turkey" dish="Dave"/>
</meal>
```

```
<!ELEMENT p (#PCDATA|a|ul|b|i|em)*>
```
p can contain text or a, ul, b, i or em elements in no particular order.

```
<!ELEMENT spec (front, body, back?)>
```
It also expresses that the spec element contains one front, one body and one optional back children elements in this order.

```
<!ELEMENT b (#PCDATA)>
```
b contains text or being of mixed content (text and elements in no particular order):

```
<!ATTLIST termdef name CDATA #IMPLIED>
```
means that the element termdef can have a name attribute containing text (CDATA) and which is optional (#IMPLIED). The attribute value can also be defined within a set:

The content type of an attribute can be text (CDATA), anchor/reference/references (ID/IDREF/IDREFS), entity(ies) (ENTITY/ENTITIES) or name(s) (NMTOKEN/NMTOKENS). The following defines that a chapter element can have an optional id attribute of type ID, usable for reference from attribute of type IDREF:

```
<!ATTLIST chapter id ID #IMPLIED>

<!ATTLIST termdef
      id      ID      #REQUIRED
      name    CDATA   #IMPLIED>

<!ATTLIST list type (bullets|ordered|glossary) "ordered">
```

means list element have a type attribute with 3 allowed values "bullets", "ordered" or "glossary" and which default to "ordered" if the attribute is not explicitly specified.

DTD Exercises 1

```
<!ELEMENT Course_Catalog (Department*)>
<!ELEMENT Department (Title,Chair,Course*)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Chair (Professor)>
<!ELEMENT Professor (First_Name,Middle_Initial?,Last_Name)>
<!ELEMENT First_Name (#PCDATA)>
<!ELEMENT Middle_Initial (#PCDATA)>
<!ELEMENT Last_Name (#PCDATA)>
<!ELEMENT Course (Title,Description?,Instructors,Prerequisites*)>
<!ELEMENT Description (#PCDATA)>
<!ELEMENT Instructors ((Lecturer|Professor)*)>
<!ELEMENT Prerequisites (Prereq+)>
<!ELEMENT Prereq (#PCDATA)>
<!ELEMENT Lecturer (First_Name,Middle_Initial?,Last_Name)>
```

```
<!ATTLIST Department Code ID #REQUIRED>
<!ATTLIST Course
  Number CDATA #REQUIRED
  Enrollment CDATA #IMPLIED>
```

DTD Exercise 2

```
<!ELEMENT Course_Catalog (Department*)>
<!ELEMENT Department (Title,Course*,(Professor|Lecturer)+)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Professor (First_Name,Middle_Initial?,Last_Name)>
<!ELEMENT First_Name (#PCDATA)>
<!ELEMENT Middle_Initial (#PCDATA)>
<!ELEMENT Last_Name (#PCDATA)>
<!ELEMENT Course (Title,Description?)>
<!ELEMENT Description (#PCDATA|Courseref)*>
<!ELEMENT Lecturer (First_Name,Middle_Initial?,Last_Name)>
<!ELEMENT Courseref EMPTY>


<!ATTLIST Department
     Code ID #REQUIRED
     Chair IDREF #REQUIRED>
<!ATTLIST Course
  Number ID #REQUIRED
  Enrollment CDATA #IMPLIED
     Instructors IDREFS #REQUIRED
     Prerequisites IDREFS #IMPLIED>

<!ATTLIST Professor
     InstrID ID #REQUIRED>

<!ATTLIST Lecturer
     InstrID ID #REQUIRED>

<!ATTLIST Courseref
  Number IDREF #REQUIRED>
```

DTD Assignment 3

```
<!ELEMENT countries (country*)>
<!ELEMENT country (language|city)*>
<!ELEMENT language (#PCDATA)>
<!ELEMENT city (name,population?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT population (#PCDATA)>

<!ATTLIST country
```

```
    name CDATA #REQUIRED
  population CDATA #REQUIRED
  area CDATA #REQUIRED
>
<!ATTLIST language
  percentage CDATA #REQUIRED
>
```

RELATIONAL ALGEBRA

RA supports the following relational algebra operators:

- \select_{*cond*} is the relational selection operator. For example, to select people with name Amy or Ben, we write "\select_{name='Amy' or name='Ben'} Person". The syntax for *cond* follows SQL: string literals can be enclosed in single or double quotes, and boolean operators and, or, and not may be used. Comparison operators <=, <, =, >, >=, and <> work on both string and numeric types.
- \project_{*attr_list*} is the relational projection operator, where *attr_list* is a comma-separated list of attribute names. For example, to find the pizzas served by Applewood (but without the price information), we write "\project_{pizza} (\select_{pizzeria='Applewood'} Serves)".
- \cross is the relational cross-product operator. For example, to compute the cross-product of Person and Frequents, we write "Person \cross Frequents".
- \join is the relational natural join operator. For example, to join Person(name,age,gender) and Frequents(name,pizzeria) enforcing equality on the shared name attribute, we simply write "Person \join Frequents". Natural join automatically equates all pairs of identically named attributes from its inputs (in this case, name), and outputs only one attribute per matching pair. The schema of the result in our example is(name,age,gender,pizzeria).
- \join_{*cond*} is the relational theta-join operator. For example, to join the two relations Person(name,age,gender) and Serves(pizzeria,pizza,price)enforcing that the pizza price is lower than the person's age, we write "Person \join_{age>price} Serves". Syntax for *cond* again follows SQL; see notes above for \select.
- \union, \diff, and \intersect are the relational union, difference, and intersection operators, respectively. As a trivial example, to compute the union between Person and itself, we write "Person \union Person;", which returns the original Person relation. To compute the difference between Person and itself, we write "Person \diff Person;", which returns the empty relation. To compute the intersection between Person and itself, we write "Person \intersect Person;", which again returns the original Person relation. **Warning:** RA allows these operators to be applied to any two subexpressions that produce an equal number of attributes, even if the corresponding attribute names don't match. (See also the note about attribute order under **Limitations** below.) This allowance is typical of most SQL implementations but violates the requirements of pure relational algebra. As good practice, and for unambiguous attribute names in the result, we suggest using the \rename operator (next) as needed to enforce matching schemas whenever \union, \diff, or \intersect is used.
- \rename_{*new_attr_name_list*} is the relational renaming operator, where *new_attr_name_list* is a comma-separated list of new names, one for each attribute of the input relation. For example, to rename the attributes of relation Person and compute the cross-product of Person with itself, we write "\rename_{name1,age1,gender1}

Person \cross \rename_{name2,age2,gender2} Person;".

Limitations

Currently, RA has the following limitations:

- \rename only supports renaming of attributes; it does not support renaming of relations.
- RA expressions may yield multiple attributes with the same name, but only in the outermost experssion -- an error may occur if you try to refer to or even rename such attributes "later" in the same expression. If a subexpression will yield multiple attributes with the same name, you should use the \rename operator within the subexpression to make the names unique.
- The standard *relName.attrName* notation for referencing an attribute is neither needed nor supported. Using the \rename operator, attribute name names can always be made unambiguous.
- As in most SQL implementations, attribute order is relevant in relations and in the result of expressions. This property is significant primarily for set operators, which as mentioned above do not consider attribute names. For example, if we take the \union of relations R(A,B) and S(B,A), the result contains two columns: (1) the union of the A values in R and the B values in S; (2) the union of the B values in R and the A values in S. As mentioned above, as good practice we suggest using the \rename operator to enforce matching schemas on R and S before applying the \union.
- Error messages in response to ill-formed RA expressions may not be especially meaningful. Recall that RA translates relational algebra expressions into SQL queries. Often an incorrect RA expression simply results in incorrect SQL queries. In these cases, RA just passes back the error messages from the underlying DBMS, without attempting to create RA-specific messages.

1)
\project_{pizza}(\select_{gender='female' and age>20}(Person \join Eats))

2)
\project_{name}(\select_{gender='female' and pizzeria='Straw Hat'}((Person \join Eats) \join Serves))

3)
\project_{pizzeria}(\select_{price<10 and (name='Fay' or name='Amy')}(Serves \join Eats))

4)
\project_{pizzeria3}(\select_{pizza1=pizza2 and pizza2=pizza3 and price3<10}((\rename_{name1,pizza1}(\select_{name='Fay'}(Eats))) \cross (\rename_{name2,pizza2}(\select_{name='Amy'}(Eats))) \cross (\rename_{pizzeria3,pizza3,price3}(Serves))))
or
\project_{pizzeria3}(\select_{name1='Fay' and name2='Amy' and pizza1=pizza2 and pizza2=pizza3 and price3<10}((\rename_{name1,pizza1}(Eats)) \cross (\rename_{name2,pizza2}(Eats)) \cross (\rename_{pizzeria3,pizza3,price3}(Serves))))

5)
\project_{name}((\select_{pizzeria='Dominos'}(Serves) \join Eats))

```
\diff
\project_{name}((\select_{pizzeria='Dominos'}(Frequents) ))

6)
(
\project_{pizza}(Eats)
\diff
\project_{pizza}(
\select_{age>=24}(Person)
\join
Eats
)
)
\union
(
\project_{pizza}(\select_{price<10}(Serves))
\diff
\project_{pizza}(\select_{price>=10}(Serves))
)

7)

\project_{age}(
\select_{pizza='mushroom'}(Eats)
\join
Person
)

\diff

\project_{age2}
(

(
\rename_{age1}(
\project_{age}(
\select_{pizza='mushroom'}(Eats)
\join
Person
)
)
)

\join_{age1>age2}

(
\rename_{age2}(
\project_{age}(
\select_{pizza='mushroom'}(Eats)
\join
Person
)
)
)

)

8)

\project_{pizzeria} (Serves)
\diff

\project_{pizzeria}(
(
\project_{pizza}(Eats)
\diff
\project_{pizza}
(\select_{age>=30}(Person) \join Eats)
)
\join
Serves
)

9)
\project_{pizzeria}(Serves)
\diff
\project_{pizzeria}
(
(
\project_{pizzeria}(Serves)
\cross
\project_{pizza}
(\select_{age>=30}(Person) \join Eats)
)

\diff

(
\project_{pizzeria,pizza}(Serves)
)
)
```

SQL
-MOVIE QUERY HOMEWORK

1/ select title from Movie where director='Steven Spielberg';

2/ select year from Movie where mID in (select mID from Rating where stars>3) order by year;

3/ select title from Movie where mID not in (select mID from Rating);

4/ select name from Reviewer where rID in (select rID from Rating where ratingDate is null);

```
5/ select name, title, stars, ratingDate
from Movie, Reviewer, Rating
where Movie.mID = Rating.mID and Reviewer.rId = Rating.rID
order by name, title, stars;

6/
select name, title
from Reviewer, Movie, (select R1.rID, R1.mID
from Rating R1, Rating R2
where R1.rId = R2.rID and R1.mID=R2.mID and R1.ratingDate < R2.ratingDate and R1.stars <
R2.stars) as G
where Reviewer.rID = G.rID and Movie.mID = G.mID;

7/
select title, G.stars
from Movie, (select mID, max(stars) as stars
from Rating
group by mID) as G
where Movie.mID = G.mID
order by title;

8/
select title, G.spread
from Movie, (select mID, max(stars) - min(stars) as spread
from Rating
group by mID) as G
where movie.mID = G.mID
order by G.spread desc, title;

9/
select distinct (
select avg(avgMovieRating) as avgBefore1980
from (
select title, year, avgMovieRating
from Movie, (select mID, avg(stars) as avgMovieRating
from Rating
group by mID) as G
where Movie.mId = G.mID and year<1980
))
-
(select avg(avgMovieRating) as avgAfter1980
from (
select title, year, avgMovieRating
from Movie, (select mID, avg(stars) as avgMovieRating
from Rating
group by mID) as G
where Movie.mId = G.mID and year>=1980
))
from Movie
;
```

SQL
- SQL Movie-Rating Modification Exercises

1/ insert into Reviewer values (209,’Roger Ebert’) ← order of values matters!

2/ insert into Rating
select R.rID, M.mID, 5, null
from Reviewer R,Movie M
where R.name = 'James Cameron';

3/
update Movie
set year = year + 25
where mID in (select mID from (select mID, avg(stars) as avgRating
from Rating
group by mID
having avgRating>=4
))

Movie (mID, title, year, director)
English: There is a movie with ID number *mID*, a *title*, a release *year*, and a *director*.

Reviewer (rID, name)
English: The reviewer with ID number *rID* has a certain *name*.

Rating (rID, mID, stars, ratingDate)
English: The reviewer *rID* gave the movie *mID* a number of *stars* rating (1-5) on a certain *ratingDate*.

SQL
- SOCIAL NETWORK QUERY

1/
select name
from Highschooler
where ID in (select ID1 ← no need to match ID and ID1 with ‘as ID’
from Friend
where ID2 in (select ID from Highschooler where name = 'Gabriel')
);

2/
select H1.name, H1.grade, H2.name, H2.grade
from Highschooler H1, Likes L, Highschooler H2
where H1.ID = L.ID1 and L.ID2 = H2.ID and H1.grade >= H2.grade +2;

3/
```
select H1.name, H1.grade, H2.name, H2.grade
from Highschooler H1, (select L1.ID1 as ID1, L1.ID2 as ID2
from Likes L1, Likes L2
where L1.ID1 = L2.ID2 and L1.ID2 = L2.ID1) as G, Highschooler H2
where H1.ID = G.ID1 and H2.ID = G.ID2 and H1.name < H2.name;
```

4/
```
select name, grade
from Highschooler
where ID not in (select ID1 from Likes union select ID2 from Likes)
```

5/
```
select H1.name, H1.grade, H2.name, H2.grade
from Highschooler H1, Likes, Highschooler H2
where ID2 in (select ID
from highschooler
where ID not in (select ID1 from Likes)
) and H1.ID = ID1 and H2.ID = ID2
```

6/
```
select name, grade
from Highschooler
where not ID in (select H1.ID
from Highschooler H1, Friend F, Highschooler H2
where H1.ID = F.ID1 and H2.ID = F.ID2 and H1.grade <> H2.grade )
order by grade, name
```

7/
```
select H1.name, H1.grade, H2.name, H2.grade, H3.name, H3.grade
from Highschooler H1, (select Likes.ID1 as Id1, Likes.ID2 as ID2
from Likes
where Likes.ID1 not in (select Friend.ID1 from Friend where Friend.ID2 = Likes.ID2)) as G,
Highschooler H2, Friend F1, Friend F2, Highschooler H3
where H1.ID = G.ID1 and H2.ID = G.ID2 and F1.ID1 = G.ID1 and F2.ID1 = G.ID2 and F1.ID2 =
F2.ID2 and H3.ID = F1.ID2
```
8/
```
select distinct (select count(*)
from Highschooler H1) - (select count (distinct name) from Highschooler)
from Highschooler
```

9/
```
select name, grade
from highschooler
where ID in ( select ID2
from Likes
group by ID2
having count(*)>=2)
```

SQL

- Social Network Modification

1/ delete from highschooler where grade=12;

2/
delete from Likes where exists (select * from Friend where Friend.ID1=Likes.ID1 and Friend.ID2=Likes.ID2) and not exists (select * from Likes L2 where L2.ID1 = Likes.ID2 and L2.ID2 = Likes.ID1);

Highschooler (ID, name, grade)
English: There is a high school student with unique *ID* and a given *first name* in a certain *grade*.

Friend (ID1, ID2)
English: The student with *ID1* is friends with the student with *ID2*. Friendship is mutual, so if (123, 456) is in the Friend table, so is (456, 123).

Likes (ID1, ID2)
English: The student with *ID1* likes the student with *ID2*. Liking someone is not necessarily mutual, so if (123, 456) is in the Likes table, there is no guarantee that (456, 123) is also present.

XPATH

1/
Return all Title elements (of both departments and courses).

doc("courses.xml")//Title

2/
Return last names of all department chairs.

doc("courses.xml")//Chair/Professor/Last_Name

3/
Return titles of courses with enrollment greater than 500.
doc("courses.xml")//Course[@Enrollment>500]/Title

4/
Return titles of departments that have some course that takes "CS106B" as a prerequisite.
doc("courses.xml")//Department[Course/Prerequisites/Prereq = "CS106B"]/Title

5/
Return last names of all professors or lecturers who use a middle initial. Don't worry about eliminating duplicates.
doc("courses.xml")//(Professor|Lecturer)[Middle_Initial]/Last_Name

6/

Return the count of courses that have a cross-listed course (i.e., that have "Cross-listed" in their description).

```
count(doc("courses.xml")//Course[contains(Description, "Cross-listed")])
```

7/

Return the average enrollment of all courses in the CS department.

```
let $plist := doc("courses.xml")//Department[@Code="CS"]/Course/@Enrollment
return avg($plist)
```

8/

Return last names of instructors teaching at least one course that has "system" in its description and enrollment greater than 100.

```
for $c in doc("courses.xml")//Course
where contains($c/Description,"system") and $c/@Enrollment > 100
return $c/Instructors/(Lecturer|Professor)/Last_Name
```

9/

Return the title of the course with the largest enrollment.

```
let $m := max(doc("courses.xml")//Course/@Enrollment)
for $c in doc("courses.xml")//Course
where $c/@Enrollment = $m
return  $c/Title
```

1/

Return a list of department titles.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:template match="Department">
<Title><xsl:value-of select="Title"/></Title>
</xsl:template>
</xsl:stylesheet>
```

2/

Return a list of department elements with no attributes and two subelements each: the department title and the entire Chair subelement structure.

```
        <xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:template match="Department">
                <Department>
                <xsl:copy-of select="Title"/>
                <xsl:copy-of select="Chair"/>
                </Department>
        </xsl:template>
        </xsl:stylesheet>
```

1/

Return the area of Mongolia.

```
doc("countries.xml")/countries/country[@name="Mongolia"]/data(@area)
```

2/

Return the names of all cities that have the same name as the country in which they are located.

```
for $c in doc("countries.xml")/countries/country
for $t in $c/city
where contains($t/name, $c/@name )
return $t/name
```

3/

Return the average population of Russian-speaking countries.

```
let $a := avg(doc("countries.xml")/countries/country[language="Russian"]/@population)
return $a
```

4/

Return the names of all countries that have at least three cities with population greater than 3 million.

```
doc("countries.xml")/countries/country[count(city[population>3000000]) > 3  ]/data(@name)
```

5/

Create a list of French-speaking and German-speaking countries. The result should take the form:

```
<result>
  <French>
    <country>country-name</country>
    <country>country-name</country>
    ...
  </French>
  <German>
    <country>country-name</country>
    <country>country-name</country>
    ...
  </German>
</result>
```

```
<result>
    <French>
         { for $c in doc("countries.xml")/countries/country[language="French"]
                  return <country> {$c/data(@name)} </country>
                  }
    </French>
    <German>
         { for $c in doc("countries.xml")/countries/country[language="German"]
                  return <country> {$c/data(@name)} </country>
                  }
```

```
   </German>
</result>
```

6/
Return the countries with the highest and lowest population densities. Note that because the "/" operator has its own meaning in XPath and XQuery, the division operator is infix "div". To compute population density use "(@population div @area)". You can assume density values are unique. The result should take the form:

```
<result>
  <highest density="value">country-name</highest>
  <lowest density="value">country-name</lowest>
</result>
```

```
<result>
   {
   let $a := max(doc("countries.xml")/countries/country/(@population div @area))
   for $c in doc("countries.xml")/countries/country
   where ($c/@population div $c/@area) = $a
   return <highest density="{$a}"> {$c/data(@name)} </highest>
   }
 {
   let $a := min(doc("countries.xml")/countries/country/(@population div @area))
   for $c in doc("countries.xml")/countries/country
   where ($c/@population div $c/@area) = $a
   return <lowest density="{$a}"> {$c/data(@name)} </lowest>
   }

</result>
```

1/
Return all countries with population between 9 and 10 million. Retain the structure of country elements from the original data.
Your solution should fill in the following stylesheet:

```
  <?xml version="1.0" encoding="ISO-8859-1"?>
   <xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
      <xsl:template match=...>
         ... template body ...
      </xsl:template>
      ... more templates as needed ...
   </xsl:stylesheet>
```

```
         <xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
           <xsl:template match="text()"/>
         <xsl:template match="country[@population &gt; 9000000 and @population &lt;
10000000]">
                  <xsl:copy-of select="."/>
```

```
</xsl:template>
</xsl:stylesheet>
```

2/
Create a table using HTML constructs that lists all countries that have more than 3 languages. Each row should contain the country name in bold, population, area, and number of languages. Sort the rows in descending order of number of languages. No header is needed for the table, but use <table border="1"> to make it format nicely, should you choose to check your result in a browser. (**Hint:** You may find the data-type and order attributes of <xsl:sort> to be useful.)

```
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
<html>
        <table border="1">
                <xsl:for-each select="countries/country">
                        <xsl:sort select="-count(language)"/>
                        <xsl:if test="count(language) &gt; 3">
                                <tr>
                <td><b><xsl:value-of select="@name"/></b></td>
                <td><xsl:value-of select="@population"/></td>
                <td><xsl:value-of select="@area"/></td>
                <td><xsl:value-of select="count(language)"/></td>
                </tr>
                </xsl:if>
                </xsl:for-each>
                </table></html>
</xsl:template>
</xsl:stylesheet>
```

3/
Create an alternate version of the countries database: for each country, include its name and population as sublements, and the number of languages and number of cities as attributes (called "languages" and "cities" respectively).

```
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
        <countries>
<xsl:for-each select="countries/country">
        <country cities="{count(./city)}" languages="{count(./language)}">
        <name><xsl:value-of select="@name"/></name>
<population><xsl:value-of select="@population"/></population>
</country>
</xsl:for-each>
</countries>
</xsl:template>
</xsl:stylesheet>
```

UML

1/
OK
Book(ISBN, edition)
Publication(ISBN, pub-title, year)
Publication(ISBN, pub-title, year, volume, edition)
Collection(ISBN,volume)
Publication(ISBN, pub-title, year, name, start-end,end-date)

NOK
Publication(name, publisher, ISBN, pub-title, year) ← start-date and end-date?
Book(edition) ← PK?
Publication(Name,ISBN)
2/

{B,R} is incomplete and overlapping (Flat liner)
{L,S} is complete and overlapping

3/

Aucune constrainte on |A| due to 0..*
Size(B) > Size(A) due to 1..1

OK
|A| = 1 , |B| = 10, |C| = 20 ← good
|A| = 0 |B| =0 |C| = 10

NOK
|B| = 10 and |C| = 1
|B| = 20 and |C| = 10
|B| = 1 and |C| = 0

4/
OK
an author may have written nothing ← good

5/
OK
Each city has at most one major ← good

6/
Edits key could be Publication.ISBN, Editor.name but because of 1..1 we can have just
Publication.ISBN ← good

7/
OK
No two countries can have the same name ← good

NOK
Each country must belong to a continent
A continent may have no countries

A country can speak 2 languages

8/
Article is the included class in a Composition relationship. Thus, its relation should contain the attributes of Article as given in this choice, plus the key of its including class which is missing from this choice.

Article(title, pages, keywords,ISBN)

NOK
Article(title, pages, keywords)

INDEX

1/

A = 1000
B = 20
C = 10
D = 10
E = 1

2/

Not useful
Course.department
Student.StudentID

Useful
Enroll.studentID
Course.studentID
INstructor,instrID

Student.studentID
Course.instrID

Given a course name, find the department offering that course.
select department
from Course
where courseName=X

List all studentIDs together with all of the departments they are taking courses in.
Select studentID, department
From Student

Given a major, return the studentIDs of students in that major.
select studentID

```
from Student
where major= ….
```

TRIGGER

1/
```
create trigger Trig1
after insert on Highschooler
when New.name = 'Friendly'
begin
        insert into Likes
        select New.ID,ID
        from highschooler
        where grade=New.grade and ID<>New.ID;
end
```

2/
```
create trigger T1
after insert on highschooler
when New.grade<9 or New.grade>12
begin
        update highschooler
        set grade=NULL
        where ID=New.ID;
end;
|                                       ← required for workbench multi trigger
create trigger T2
after insert on highschooler
when New.grade is NULL
begin
        update highschooler
        set grade=9
        where ID=New.ID;
end;
```

3/
```
create trigger T1
after delete on friend
begin
        delete from friend
        where ID1=Old.ID2 and ID2=Old.ID1;
end;
|
create trigger T2
after insert on friend
```

```
begin
        insert into friend values (New.ID2,New.ID1);
end;
```

4/

```
create trigger T1
after update on highschooler
when New.grade>12
begin
        delete from highschooler
        where ID=New.ID;
end;
```

5/

```
create trigger T1
after update on highschooler
when New.grade>12
begin
        delete from highschooler
        where ID=New.ID;
end;
|
create trigger T2
after update on highschooler
begin
        update highschooler
        set grade=New.grade
        where grade=Old.grade and grade=New.grade-1;
end;
```

VIEWS

(1)

```
create view LateRating as
  select distinct R.mID, title, stars, ratingDate
  from Rating R, Movie M
  where R.mID = M.mID
  and ratingDate > '2011-01-20'

create trigger LateRatingUpdate
instead of update of title on LateRating          ← instead of trigger on view
for each row
when New.mID in (select mID from Rating where ratingDate > '2011-01-20')
```

```
begin
update Movie
set title = New.title
where mID = Old.mID;
end;
```