EXAM
Databases (DIT620/TDA355/TDA356/TDA357)

DAY: 9 Mar 2010 TIME: 8:30 – 12:30 PLACE: Hörsalsvägen

Responsible:	Niklas Broberg, Computing Science mobil 0706 49 35 46
Results:	Will be published on the course web page after the exam
Extra aid:	A single, hand-written A4 paper. It is legal to write on both sides. This paper should be handed in with the exam.
Grade intervals:	**U**: 0 – 23p, **3**: 24 – 35p, **4**: 36 – 47p, **5**: 48 – 60p, **G**: 24 – 41p, **VG**: 42 – 60p, **Max.** 60p.

IMPORTANT

The final score on this exam is computed in a non-standard way. The exam is divided into 7 blocks, numbered 1 through 7, and each block consists of 2 or 3 levels, named A, B, and optionally C. A level can contain any number of subproblems numbered using i, ii and so on. In the final score you can only count **ONE** level from each block. For example: if you attempt to solve the problems on all three levels in block 4 and manage to obtain 4 points for 4A (block 4, level A), 1 point for 4B and 8 points for 4C, only problem 4C (where you got your highest score) will count towards your final result, so your score for block 4 will be 8 points.

The score for each problem depends on how difficult it is (more points for harder problems) and how important I think it is (more points for more important problems). It does *not* depend on how much work it takes to answer the problem. There could very well be a 12 point problem that takes 15 seconds to answer (given that you know the right answer, of course).

The problems in each block are ordered by increasing difficulty. Hence the A problems are easy, but aim to cover the full basics of its area. The B and C level problems are more difficult, and aim to test your knowledge of the areas beyond the mere basics. If you only solve A problems your maximum score is 35 points, and if you only solve the B problems where there are also C problems it is 38 points.

Please observe the following:

- Answers can be given in Swedish or English
- Use page numbering on your pages
- Start every assignment on a fresh page
- Write clearly; unreadable = wrong!
- Fewer points are given for unnecessarily complicated solutions
- Indicate clearly if you make assumptions that are not given in the assignment

Good advice

- Most problems have been designed to give short answers. Few problem should require more than one page to answer.
- There are more problems than you are likely to solve in 4 hours. This means that you have to think about which problems you attempt to solve. If you try solve the problems in the order they are given, **you are likely to fail the exam**!

Good Luck!

Block 1 - Entity-Relationship Diagrams **max 12p**

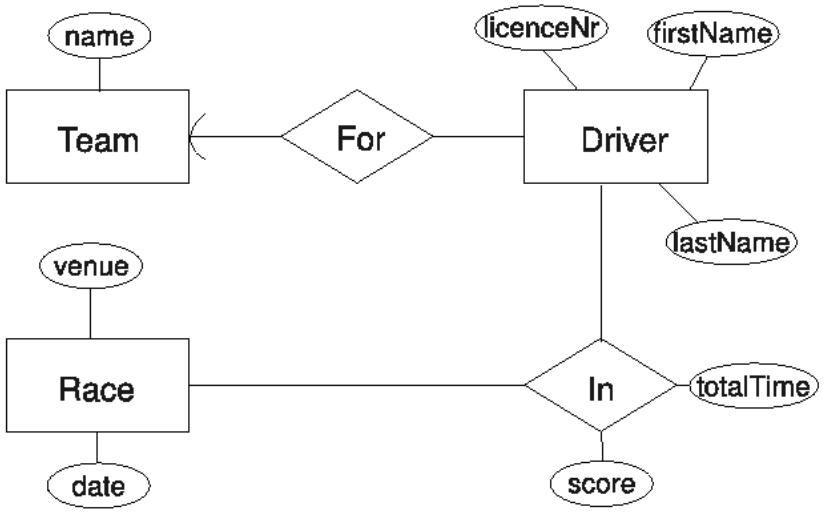
1A (8p)

(i) (4p)

A small racing league want a database to keep track of teams, drivers, races and scores in the league. The league is run for teams, which are identified by their names. Each team has one or more drivers signed up, and each driver is registered with the league and has a unique league licence number. First and last names of the drivers should also be included. A driver may only participate for a single team throughout the season. Races are identified simply by the dates when they are run. For each race, the league also wants to store the venue where it took place. Drivers participate in races, and for each participating driver the database should store the total race time for that driver, and the league score they got from that race.

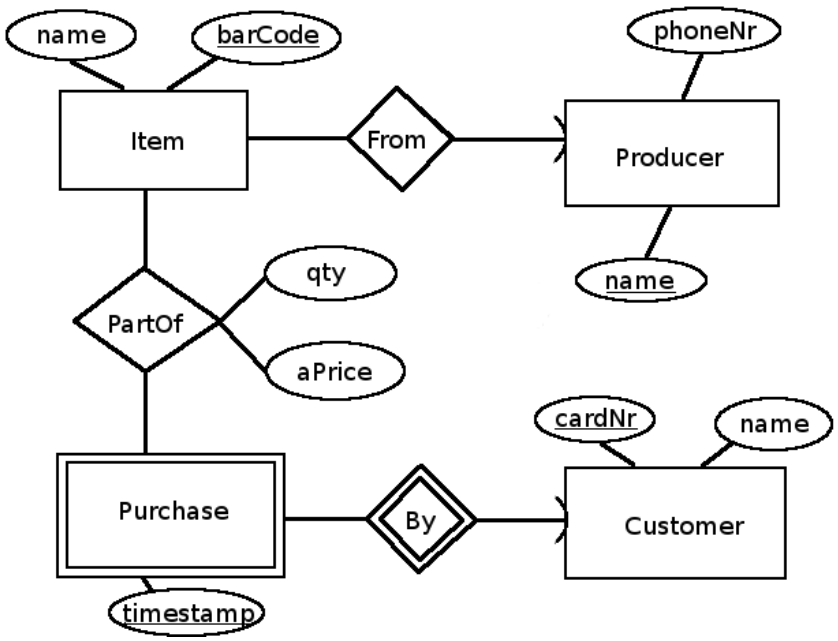
Your task is to draw an ER diagram that correctly models this domain and its constraints.

Solution:



(ii) (4p)

The E/R diagram below is part of an E/R diagram for a database used by a store to keep track of its sales.



Translate this ER diagram into a set of relations. Mark keys and references clearly in your answer.

Solution:

Producers(<u>*name*</u>, *phoneNr*)
Items(<u>*barCode*</u>, *name*, *producer*)
    producer → *Producers.name*
Customers(<u>*cardNr*</u>, *name*)
Purchases(<u>*customer*</u>, <u>*timestamp*</u>)
    customer → *Customers.cardNr*
PartOf(<u>*item*</u>, <u>*customer*</u>, <u>*purchase*</u>, *qty*, *aPrice*)
    item → *Items.barCode*
    (*customer*, *purchase*) → *Purchases.*(*customer*, *timestamp*)

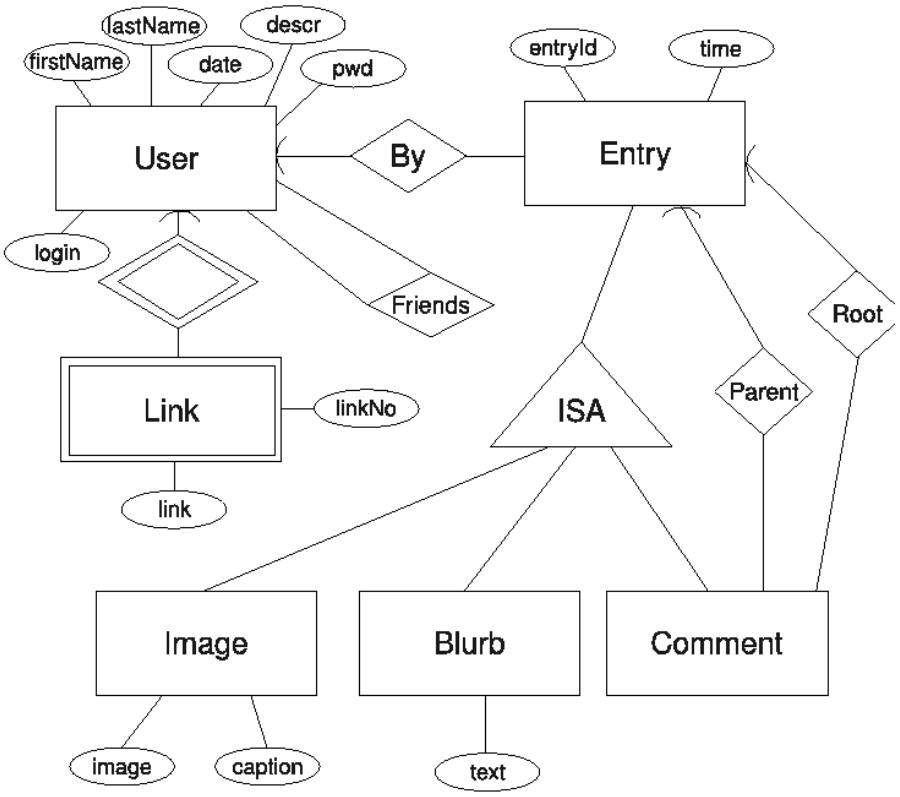
1B (12p)

Below is a database schema used for a social networking web site:

Users(*login*, *firstName*, *lastName*, *password*, *description*, *dateJoined*)
Links(*user*, *linkNo*, *link*)
    user → *Users.login*
Entries(*entryId*, *user*, *time*)
    user → *Users.login*
Comments(*entry*, *parentEntry*, *rootEntry*, *text*)
    entry → *Entries.entryId*
    parentEntry → *Entries.entryId*
    rootEntry → *Entries.entryId*
Images(*entry*, *caption*, *image*)
    entry → *Entries.entryId*
Blurbs(*entry*, *text*)
    entry → *Entries.entryId*
Friends(*user*, *friend*, *sinceDate*)
    user → *Users.login*
    friend → *Users.login*

All of this should be self-explanatory (or part of the problem), but if there is something that you don't understand, don't hesitate to ask. (Incidentally this is the same schema used in block 3 and onwards. A fuller explanation of the domain is given there, but won't really help with this particular problem.)
Reconstruct the ER diagram that led to these relations and constraints.

Solution:


lastName
descr
firstName
date
pwd
entryId
time
User
By
Entry
login
Friends
Root
Link
linkNo
ISA
Parent
link
Image
Blurb
Comment
image
caption
text

Block 2 - Dependencies and Normal Forms **max 12p**

2A (8p)

A small banking enterprise uses a database to manage the accounts of their clients. The bank is divided into local branches, identified by unique branch codes. Accounts belong to a specific branch, and are identified by account numbers that are unique within that branch. An account can be shared by several clients, and the only other information necessary to store is the account balance.

For clients, the bank stores their social security number, first and last names, as well as a contact address and phone number.

The schema they use for their database is quite simple, and looks like this:

Clients(*ssNr*, *firstName*, *lastName*, *address*, *phoneNumber*)
Accounts(*branchCode*, *branchName*, *accountNr*, *balance*, *client*)
client → *Clients.ssNr*

This schema is not fully normalized, and thus suffers from a number of problems. It is your task to solve these by normalization of the schema.

(i) (4p)
For the given domain, identify all functional dependencies that are expected to hold.

Solution:

(1) $ssNr \rightarrow firstName, lastName, address, phoneNumber$
(2) $branchCode \rightarrow branchName$
(3) $branchCode, accountNr \rightarrow balance$

(ii) (1p)
With the dependencies you have found, identify all BCNF violations in the relations of the database. For each violation, also specify what kinds of problems that could arise if it was not resolved.

Solution: (1) above is not a violation. Both (2) and (3) violate BCNF on Accounts.

(iii) (3p)
Do a complete normalization of the schema, so that all relations are in BCNF. (It's the end product that's important, not the steps you take to get there.)

Solution:

Clients(*ssNr*, *firstName*, *lastName*, *address*, *phoneNumber*)
Branches(*branchCode*, *branchName*)
Accounts(*branchCode*, *accountNr*, *balance*)
    branchCode → *Branches.branchCode*
AccountOwners(*branchCode*, *accountNr*, *client*)
    (*branchCode*, *accountNr*) → *Accounts.*(*branchCode*, *accountNr*)
    client → *Clients.ssNr*

2B (12p)

A library stores information about books and authors, among other things that we disregard for this problem. We assume for simplicity that books can be uniquely identified by their title, and authors by their name. For books, the database also stores the year it was first published. A book *version* is a version of a book in a specific language, on a specific media (paper version, audio book, digital version, etc). For each version, the database stores its unique *ISBN number*, as well as the media and language of the version. Book versions are also categorized, for easy indexing. There are a lot of categories a book version can be tagged with (e.g. fiction, geography, child novel, short version, etc), and a book can have any number of categorizations. Note specifically that it's the book versions that are categorized, so different versions of the same book need not be tagged with the same set of categories.

The following relation sums up all the attributes that should be stored in the database:

Library(*authorName*, *bookTitle*, *yearPub*, *ISBN*, *media*, *language*, *category*)

Your task is to use normalization techniques to find a suitable schema for this database.

(i) (8p)
Find all dependencies and independencies (aka multi-valued dependencies) that are expected to hold for this domain given the domain description above.

Solution:

ISBN → *media*, *language*
bookTitle → *yearPub*

bookTitle ↠ *authorName*
ISBN ↠ *category*

(ii) (4p)
Do a complete decomposition of *Library* so that the resulting schema fulfills 4NF.

Solution:

$Books(\underline{bookTitle}, yearPub)$
$BookAuthors(\underline{bookTitle}, \underline{bookAuthor})$
$\quad bookTitle \rightarrow Books.bookTitle$
$BookVersions(\underline{ISBN}, bookTitle, media, language)$
$\quad bookTitle \rightarrow Books.bookTitle$
$VersionCategories(\underline{ISBN}, \underline{category})$
$\quad ISBN \rightarrow BookVersions.ISBN$

If you created separate, single-attribute relations for Authors and Categories that's of course fine too, though they won't appear through the 4NF decomposition algorithm.

Block 3 - SQL Data Definition Language **max 8p**

The domain for this block, and for several following blocks as well, is that of a database for a simple social networking web site. The database keeps tracks of registered users and their registered friends. Users may post short text blurbs, or images, and all users can comment on the posts or comments of other users. Further, users may write a descriptive text about themselves on their personal page, and may also add one or more links to other sites.

You are given the following schema of their intended database:

Users(*login*, *firstName*, *lastName*, *password*, *dateJoined*, *description*)
    length(password) ≥ 8
Links(*user*, *linkNo*, *link*)
    user $\rightarrow$ *Users.login*
Entries(*entryId*, *user*, *time*)
    user $\rightarrow$ *Users.login*
Comments(*entry*, *parentEntry*, *rootEntry*, *text*)
    entry $\rightarrow$ *Entries.entryId*
    parentEntry $\rightarrow$ *Entries.entryId*
    rootEntry $\rightarrow$ *Entries.entryId*
Images(*entry*, *caption*, *image*)
    entry $\rightarrow$ *Entries.entryId*
Blurbs(*entry*, *text*)
    entry $\rightarrow$ *Entries.entryId*
Friends(*user*, *friend*, *sinceDate*)
    user $\rightarrow$ *Users.login*
    friend $\rightarrow$ *Users.login*

The first thing in this schema that requires some extra attention is comments. Images, text blurbs and comments are all considered entries, and all entries may be commented. The parent entry of a comment is simply the entry that was commented. The root entry of a comment is the image or text blurb under which the comment is written, regardless of whether the comment is written directly on that root entry, or if it is a comment to some other comment under that root entry. Comments under a given root entry thus form a tree structure, all with the same root entry.

The second thing is the Friends relation. For a given user and friend, they are only listed once, with one of them as user and the other as friend. To find all the friends of a particular user, one would thus need to look for that user both in the user and friend positions.

3A (4p)

Write SQL DDL code that correctly implements these relations as tables in a relational DBMS. Make sure that you implement all given constraints correctly. Do not spend too much time on deciding what types to use for the various columns. We will accept any types that are not obviously wrong. Don't forget to implement all specified constraints, including checks. You may assume a function LENGTH that returns the length of a string.

```
CREATE TABLE Users (
  login VARCHAR(20) PRIMARY KEY,
  firstName VARCHAR(30), lastName VARCHAR(30),
  password CHAR(20) CHECK (LENGTH(password) >= 8),
  dateJoined DATE,
  description VARCHAR(250) );

CREATE TABLE Links (
  user REFERENCES Users,
  linkNo INT,
  link VARCHAR(40),
  PRIMARY KEY (user, linkNo) );

CREATE TABLE Entries (
  entry INT PRIMARY KEY,
  user REFERENCES Users,
  time TIMESTAMP );

CREATE TABLE Comments (
  entry PRIMARY KEY REFERENCES Entries,
  parentEntry REFERENCES Entries,
  rootEntry REFERENCES Entries,
  text VARCHAR(250) );

CREATE TABLE Images (
  entry PRIMARY KEY REFERENCES Entries,
  caption VARCHAR(100),
  image BINARY );

CREATE TABLE Blurbs (
  entry PRIMARY KEY REFERENCES Entries,
  text VARCHAR(250) );

CREATE TABLE Friends (
  user REFERENCES Users,
  friend REFERENCES Users,
  sinceDate DATE,
  PRIMARY KEY (user, friend) );
```

3B (8p)

All comments made under the same root entry must also specify that same entry as its root entry. Specifically, if a comment is made on another comment, those two comments must have the same root entry. Formally, for all comments in the database, we must have either that their parent and root entries are the same entry, or that their root entry is the same as the root entry of their parent entry.

(i) (6p)
Implement this constraint as a trigger, ensuring that the constraint is not violated when new comments are added.

Solution:

```
CREATE TRIGGER ConsistentRoots
BEFORE INSERT ON Comments
FOR EACH ROW
REFERENCING NEW ROW AS newrow
WHEN (newrow.parentEntry != newrow.rootEntry
     OR (SELECT rootEntry
         FROM Comments
         WHERE entry = newrow.parentEntry) != newrow.rootEntry)
ROLLBACK;
```

Another solution could be to redirect via a view, and only insert when the root entry is consistent, that would remove the need for ROLLBACK. Using a PSM exception instead of ROLLBACK is of course also fine.

Yet another potential solution is to force consistency by overwriting the root entry in case it is inconsistent. You would then have to set the root entry in newrow to that of the parent entry.

(ii) (2p)
Discuss briefly the differences between implemeting this constraint using a trigger, and implementing it using an assertion. What are the benefits and drawbacks of either approach?

Solution: If comments are only ever added (or removed) but never changed, using a trigger is fine. The trigger will only run on insertions, and will then ensure that all comments in the database behave properly.

If comments can also be updated (in particular if their parent or root entries can be updated) then it would be possible to end up with inconsistencies, as the trigger only runs on insert. Using an assertion would ensure that the database is *never* inconsistent, but at the cost of potentially being checked on every modification to the database.

Block 4 - SQL Queries **max 8p**

Use the relations for the social networking web site from the previous block when answering the following problems.

4A (4p)

(i) (1p)
Write an SQL query that lists all comments together with the logins of the users that posted them.

Solution:

```
SELECT entryId, user
FROM Entries, Comments
WHERE entry = entryId;
```

(ii) (1p)
List all users in the order that they joined the site. The first user to join should be listed first.

Solution:

```
SELECT login, dateJoined
FROM Users
ORDER BY dateJoined;
```

(iii) (2p)
For each user, list the number of friends that user has. Remember to count both the rows where the user appears as user and when they appear as friend in the relation.

Solution:

```
SELECT user, COUNT(friend)
FROM ((SELECT user, friend FROM Friends) UNION
      (SELECT friend AS user, user AS friend FROM Friends))
GROUP BY user;
```

It's not necessary to rename the attributes in the second operand of the union as I do here.

4B (6p)

Write a query that lists all users together with the number of non-comment entries they have posted.

Solution:

```
SELECT login, (SELECT COUNT(entry)
               FROM ((SELECT entry FROM Entries WHERE user = login)
                     MINUS (SELECT entry FROM Comments)))
FROM Users;
```

The correlated query here arises since we want to list all users, even those that haven't made any non-comment entries. A different approach could be to first list the users *with* non-comment entries, and then explicitly add all other users together with 0.

```
WITH
 U AS (SELECT user, COUNT(entry) as entries
       FROM Entries, (SELECT entry FROM Images) UNION (SELECT entry FROM Blurbs)
       WHERE entryId = entry
       GROUP BY user)
(SELECT user, entries
 FROM U)
UNION
(SELECT login, 0
 FROM Users
 WHERE login NOT IN (SELECT user FROM U));
```

4C (8p)

Write a query that for each user lists that user's "best friend". This is defined to be the friend whose (parent) entries the user has commented the most times.

Solution:

```
WITH
 UFriends AS ((SELECT user, friend FROM Friends) UNION
              (SELECT friend as user, user as friend FROM Friends)),
 UFriendCom AS (SELECT UF.user, UF.friend, COUNT(*) AS nrComments
                FROM UFriends UF, Entries E1, Entries E2, Comments C
                WHERE UF.user = E1.user AND E1.entryId = C.entry
                  AND C.parentEntry = E2.entryId AND UF.friend = E2.user
                GROUP BY UF.user, UF.friend)
SELECT user, friend AS bestFriend
FROM UFriendCom UF
WHERE nrComments = (SELECT MAX(nrComments)
                    FROM UFriendCom
                    WHERE user = UF.user);
```

The first subquery just ensures that we're looking at all a user's friends, both those where the user appear as 'user' and where they appear as 'friend'. The second subquery counts the number of comments a user has done on each friend's entries.

Block 5 - Relational Algebra **max 6p**

Use the relations for the social networking web site from the previous blocks when answering the following problems.

5A (3p)

(i) (2p)
What does the following relational algebra expression compute (answer in plain text):

$\gamma_{user,COUNT(*)}(\sigma_{rootEntry=entryId}((\sigma_{entryId=entry}(Images \times Entries)) \times Comments))$

Solution: For each user, count the total number of comments made under that user's images.

(ii) (1p)
Translate the following relational algebra expression to SQL:

$\tau_{dateJoined}(\delta(\pi_{login,firstName,lastName,dateJoined}(\sigma_{login=user}(Users \times Entries))))$

Solution:

```
SELECT DISTINCT login, firstName, lastName, dateJoined
FROM Users, Entries
WHERE login = user
ORDER BY dateJoined;
```

5B (4p)

Translate the following SQL query to relational algebra:

```
SELECT user, COUNT(friend) AS friends
FROM Friends
GROUP BY user
HAVING friends >= 100
```

Solution: $\sigma_{friends>=100}(\gamma_{user,COUNT(friend)\rightarrow friends}(Friends))$

5C (6p)

Write a relational algebra expression that for each user lists the first image they posted, if any (you may disregard users that haven't posted any images).

Solution:

$R0 := \sigma_{entryId=entry}(Images \times Entries)$
$R1 := \gamma_{user,MIN(timestamp)\rightarrow first}(R0)$
$R := \sigma_{R1.user=R0.user\&R0.timestamp=R1.first}(R1 \times R0)$

Block 6 - Transactions **max 6p**

Use the relations for the social networking web site from the previous blocks when answering the following problems.

6A (3p)

Consider the following program (in pseudo-code), for allowing new users to register with the site. In the code I prefix program variables with : just to distingush them from attributes (i.e. you don't need to worry about any connection to PSM or the like).

```
1 ... get a suggested login from the user, in variable newLogin ...
2 exists := SELECT COUNT(*) FROM Users
            WHERE login = :newLogin;
3 if (exists == 1)
   ... give an error message and start over ...
  else
4   ... get fName, lName, pwd from user ...
5 INSERT INTO Users VALUES
   (:newLogin, :fName, :lName, :pwd, TODAY())
```

(i) (1p)
For the program specified above, what atomicity problems could arise if it was not run as a transaction?

Solution: Since there is only one modification, no atomicity problems could arise.

(ii) (2p)
For the program specified above, what isolation problems could arise if it was not run as a serializable transaction?

Solution: The problem that could arise is that if two instances of the program were run in parallel, they could both find the same new login to be free, and the second one to get to the insertion at (5) would get an error (from trying to violate the key constraint on logins).

6B (6p)

Compare what would happen if the program above was run as a transaction with isolation level SERIALIZABLE compared to if it was run with isolation level READ COMMITTED. Point out benefits and drawbacks of the two choices for this particular problem.

Solution: If the program was run as SERIALIZABLE, no other program would be allowed to modify the Users table between the read (2) and the write (5), so the insertion at (5) could never fail. On the other hand, the program waits for user input at (4), so any other program waiting to run could be forced to wait for a long time (indefinitely in theory).

If the program was run as READ COMMITTED, we would still suffer from the problem mentioned in A(ii) above. On the other hand no concurrent runs of the program would have to wait, and the failure is perhaps not so crucial, so this might still be the preferred option.

Block 7 - Semi-structured Data and XML **max 8p**

A small car selling enterprise has a small database over cars in stock, and store this using XML. The database they use is described by the following DTD:

```
<!DOCTYPE CarsNStars [

<!ELEMENT CarsNStars (Brand*, Car*)>
<!ELEMENT Brand (Model*)>
<!ELEMENT Model  (Year+)>
<!ELEMENT Year   (#EMPTY)>
<!ELEMENT Car (#EMPTY)>

<!ATTLIST Brand
    name    ID #REQUIRED
    country CDATA #IMPLIED>
<!ATTLIST Model
    name    ID #REQUIRED>
<!ATTLIST Year
    year       CDATA #REQUIRED
    horsePower CDATA #IMPLIED>

<!ATTLIST Car
    regNr ID    #REQUIRED
    model IDREF #REQUIRED
    miles CDATA #REQUIRED>

]>
```

(This is of course a gross simplification of reality, for instance where all cars of a given model issued a given year have the same horsepower, but that doesn't matter for our purposes here.)

7A (5p)

(i) (2p)
Give a minimal XML document that is valid with respect to the DTD above, and that contains information about at least one car. Note that minimal means as few nodes as possible, the size of the data in the nodes (e.g. the length of a brand name) is irrelevant.

Solution:

```
<CarsNStars>
 <Car regNr="ABC123" model="ABC123" miles="20000" />
</CarsNStars>
```

Yeah, that's a minimal document alright. I didn't even consider that possibility when writing the exam, it was pointed out to me as I answered questions, but it's certainly correct. The IDREF points to an ID, which is all that's needed. But I will also accept a solution like:

```
<CarsNStars>
 <Brand name="Volvo />
 <Car regNr="ABC123" model="Volvo" miles="20000" />
</CarsNStars>
```

Don't be fooled by the attribute name model here, there's nothing that says that it has to point to a Model element specifically (even if that was perhaps the actual intention of the design, and the way the data in their "real" database is actually stored).

Technically you should also have a document header (`<?xml ...`) but I won't be picky with that when correcting.

(ii) (3p)
Write an XPath expression that lists all models that have a version issued in 2010.

Solution:

```
//Model[/Year/@year = 2010]
```

7B (8p)

Write an XQuery expression that returns, for each brand, the number of cars of that brand that are listed in the database. (The function COUNT applied to a set returns the number of elements in that set.) The result should be on the form:

```
<Brand name="Volvo">53</Brand>
<Brand name="SAAB">28</Brand>
...
```

Solution

```
FOR $b IN //Brand
LET $cars :=
    (FOR $c IN //Car
     LET $cb := $c/@model => //Model/..
     WHEN $cb/@name = $b/@name
     RETURN $c)
LET $nr := COUNT($cars)
RETURN <Brand name={$b/@name}>{$nr}</Brand>
```