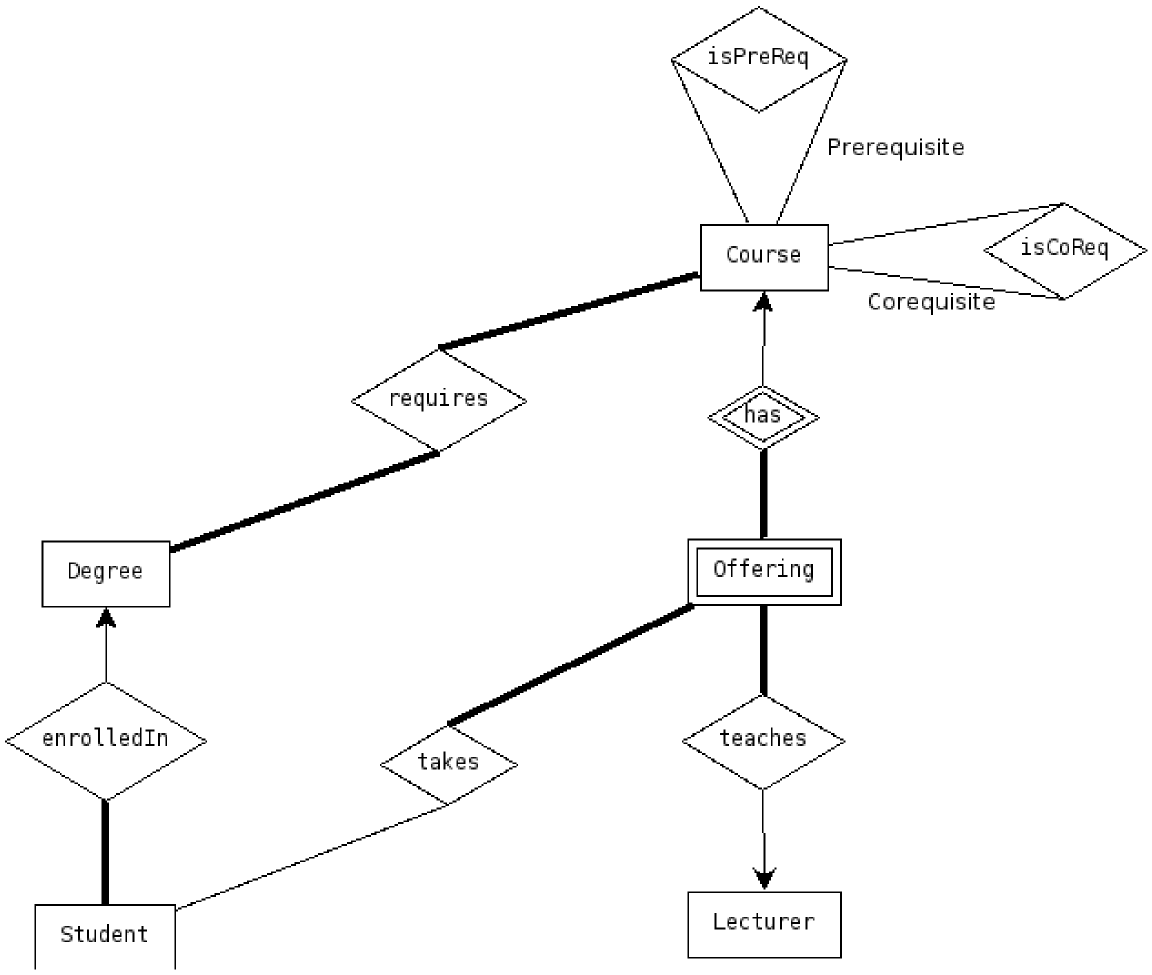
COMP9311 17s2 Exam
Sample Solution

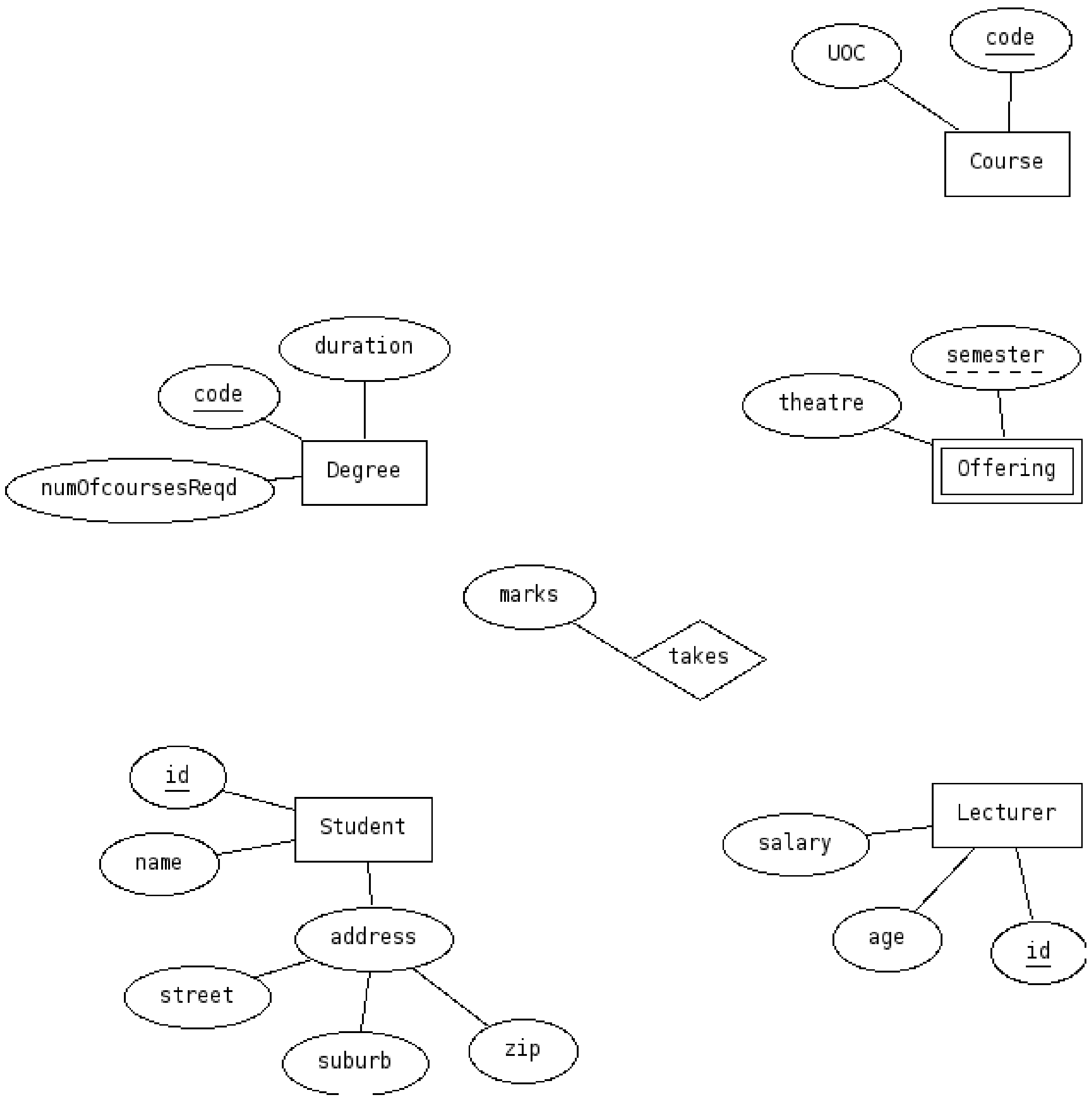
Note: There are more than one possible solutions for each question.

Q1 (12 marks)

A sample solution:



Note: You cannot relate Student and Lecturer directly with Course entity.



5 marks for the 5 entities (Degree, Course, Offering, Student, Lecturer) or similar with correct attributes as well as mentioning that Offering is a weak entity.

7 marks for the 7 relationships (requires, enrolledIn, teaches, takes, has, isPreReq, isCoReq) or similar with correct participation (partial or total) and cardinality (one-to-many or many-to-many).

Q2 **(12 marks)**

(a) Convert the ER Diagram into relational model **(4 marks)**

```
Hospital ( name (PK), address, num_of_wards )

Doctor ( name (PK), speciality, worksIn (FK references hospital(name)) )

Patient ( name (PK), date_of_birth, family_doc (FK references doctor(name)) )

Visits ( patient (PK, FK references patient(name)), doctor (PK, FK references
doctor(name)) )
```

1 mark for each correctly mapped relation (Hospital, Doctor, Patient, Visits)

(b) Write SQL statements to create tables for the ER diagram **(8 marks)**

```
create table hospital(
        name varchar(20) primary key,
        address varchar(50),
        num_of_wards integer
);

create table doctor(
        name varchar(20) primary key,
        speciality varchar(20),
        worksIn varchar(20) not null references hospital(name)
);

create table patient(
        name varchar(20) primary key,
        date_of_birth date,
        family_doc varchar(20) references doctor(name)
);


create table visits(
        patient varchar(20) references patient(name),
        doctor varchar(20) references doctor(name),
        primary key (patient, doctor)
);
```

Note: worksIn attribute is set to **NOT NULL** because the participation of Doctor in worksIn relationship is **total**. The order in which tables are created is also important. For instance, if you try to create doctor table before hospital table, you cannot add foreign key constraint (you will have to add it after creating hospital table). You may also add a constraint so that num_of_wards is never less than zero.

2 marks for each correctly created relation (Hospital, Doctor, Patient, Visits) in the correct order as well as setting worksIn to not null. Deduct 0.5 mark if worksIn is not set to not null.

Q3 (12 marks)

(a) Write relational algebra expressions for the following queries (6 marks):

(i) For each author, display the number of fans (the readers who have listed him as their favorite authors). Note: some authors may have zero fans (3 marks):

```
AuthorsFans = Proj[Author.name,Reader.name](Author Left Outer Join
[Author.name=Reader.favAuthor] Reader)

Result = GroupBy[author]count[reader](AuthorsFans)

// AuthorsFans contains the authors and their fans. An author who doesn't
// have any fan, contains NULL for reader column

// Result returns the count of reader for each fan.
// Note that if you use count(*), it is wrong because count(*) will return 1
// for the authors with reader as NULL. count(reader) ignores the tuples
// where reader value is NULL
```

(ii) Find the name of every book such that at least two readers who like the author of the book have given it a rating less than 5 (3 marks):

```
Temp = Proj[Readings.book,Readings.reader]Sel[Readings.book=Book.name AND
Book.author=Reader.favAuthor AND Reader.name=Book.name AND rating < 5](Readings X Book X
Reader)

Result = Proj[book]Sel[count>1](GroupBy[book]count[reader](Temp))

// Temp stores the (book, reader) pairs such that the book is written by the
// favorite author of the reader and reader had given it a rating less than 5

//Result projects the name of books that have more than one such readers
```

(b) Write SQL queries for the following (6 marks):

(i) For each book, display the number of its editions (3 marks):

```
select bookName, count(*) as numEditions from edition group by bookname;
```

(ii) Find the name of publishers for book(s) with greatest number of editions (3 marks):

```
-- NumEditionsBooks stores the number of editions for each book

create or replace view NumEditionsBooks as
select bookName, count(*) as numEditions from edition group by bookname;


-- selects the publishers for the book with greatest number of editions

select distinct a.publisher from author a, book b, NumEditionsBooks n
where a.name=b.author AND b.name=n.bookName AND
n.numEditions = (select max(numEditions) from NumEditionsBooks);
```

Q4 (12 marks)

(a) Write a **function** that takes the name of a reader as a parameter and prints the names of all authors living in the same city as that of the readers. If there is no such author, the function should return "**No such author!**". If the reader does not exist in our database, an exception must be raised with an appropriate message **(4 marks)**:

```
create or replace function sameCity(rname text) returns setof text
as $$
declare
     tot integer;
     rcity text;
begin
     select count(*) into tot from reader where name=rname;
     if tot=0 then
        raise exception 'Invalid reader %',rname;
     end if;
     select count(*) into tot from author a, reader r
        where r.name=rname and r.city=a.city;
     if tot=0 then
        raise debug 'No such author!';
     else
        for rec in (select a.name from author a, reader r
        where r.name=rname and r.city=a.city) loop
           raise debug 'Author %',rec.name;
           -- also give marks if student used 'return next rec.name'
        end loop;
     end if;
end;
$$ language plpgsql;
```

(b) Write a **function** that takes the name of a reader as a parameter and returns the number of books she/he has read (the **Readings** table records the books each reader has read). If the reader does not exist in the **Reader** table, **-1** must be returned **(4 marks)**:

```
create or replace function booksRead(rname text) returns integer
as $$
declare
     tot integer;
begin
     select count(*) into tot from reader where name=rname;
     if tot=0 then
        return -1;
     end if;
     select count(*) into tot from readings where reader=rname;
     return tot;
end;
$$ language plpgsql;
```

(c) Write a **trigger** such that on any insert or update on the Readings table, it checks that the name of book and the name of reader are valid (the name of book and reader exists in the Book and Reader tables, respectively). The trigger must also ensure that the value of rating is valid. More specifically, a NULL value for the rating is allowed but any non-NULL value must be between zero and ten **(4 marks)**:

```
create or replace trigger trig before insert or update on readings for each row
as $$
declare
     tot integer;
begin
     select count(*) into tot from book where name=new.book;
     if tot=0 then
       raise exception 'Invalid book %',new.book;
     end if;

     select count(*) into tot from reader where name=new.reader;
     if tot=0 then
       raise exception 'Invalid reader %',new.reader;
     end if;

     if new.rating<0 OR new.rating>10 then
       raise exception 'Invalid rating %',new.rating;
     end if;
end;
$$ language plpgsql;
```

Note that we don't need to check the old values in the trigger because insert trigger ensures that any values present in the table are valid.

Q5 (12 marks)

(a) List all the functional dependencies in the Autograph database. Include only the dependencies that are apparent from the ER diagram. Write down any assumptions you make **(5 marks)**:

(1)	TName → coach	(0.5 mark)
(2)	SName → capacity	(0.5 mark)
(3)	PName, shirtNo → TName, ranking	(1 mark)
(4)	FName → city, age	(1 mark)
(5)	FName, PName, shirtNo → SName, date	(1 mark)
(6)	FName, SName → PName, shirtNo, date	(1 mark)

All other dependencies can be derived from the above dependencies.

(b) Generate a lossless decomposition of the given table into **BCNF**. Use normalization theory and show all important steps. After decomposition, determine all candidate keys of every table **(7 marks)**:

We decompose the relation using the above set of functional dependencies.

R = { TName, coach, PName, shirtNo, ranking, SName, capacity, date, FName, city, age }

The dependency `TName → coach` violates BCNF because TName is not a key of R. We decompose R into R1 and R2 by using this dependency.

R1 = { TName, coach }
R2 = { TName, PName, shirtNo, ranking, SName, capacity, date, FName, city, age }

R1 is in BCNF since it has two attributes. Now we check R2. The dependency `PName, shirtNo → TName, ranking` violates BCNF because (PName, shirtNo) is not the key for R2 (since its closure does not include all the attributes in R2). We decompose R2 into R3 and R4 by using this dependency.

R3 = { TName, PName, shirtNo, ranking }
R4 = { PName, shirtNo, SName, capacity, date, FName, city, age }

R3 is in BCNF because the only dependency that holds on it is `PName, shirtNo ⟶ TName, ranking` and (PName, shirtNo) is the key of R3.
Now we check R4. The dependency `SName → capacity` violates BCNF. We decompose R4 into R5 and R6 using this dependency.

R5 = {SName, capacity}
R6 = {PName, shirtNo, SName, date, FName, city, age}

R5 is in BCNF since it has two attributes. However, the dependency `FName ⟶ city, age` violates BCNF in relation R6. We decompose R6 into R7 and R8.

R7 = {FName, city, age}
R8 = {PName, shirtNo, SName, date, FName}

R7 is in BCNF since FName is the key.
Now let's check R8. The dependencies that hold on R8 are:
`FName, PName, shirtNo → SName, date` and
`FName, SName → PName, shirtNo, date`

Left hand sides of both dependencies are keys for relation R8. Hence, **R8 is in BCNF**.

After decomposition into BCNF, we get the tables R1, R3, R5, R7 and R8. Below we write the tables with appropriate names and determine all possible candidate keys.

TEAM = R1 = { TName, coach }

The only candidate key is **TName** since the closure of coach does not include TName. Moreover (TName, coach) cannot be a key because it is not minimal.

PLAYER = R3 = { TName, PName, shirtNo, ranking }

The only candidate key is **(PName, shirtNo)**. No other subset of R3 determines all attributes.

STADIUM = R5 = { SName, capacity }

The only candidate key is **SName**.

FAN = R7 = { FName, city, age }

The only candidate key is **FName** because its closure includes both the city and age.

AUTOGRAPH = R8 = { PName, shirtNo, SName, date, FName }

Now we try to find all possible candidate keys for AUTOGRAPH table. None of these attributes alone form a key (check the closures). Also, since `date` does not appear on the left hand side of any dependency, it can never be part of any key.

We try other possible sets. Let us try (PName, SName, FName, shirtNo). We find the closure and it is a superkey. Is it minimal? Let’s try (PName, SName, FName). The closure includes all attributes, so this is also a superkey. But is it minimal? Let’s try (PName, SName). Its closure does not include FName and date so it is not a superkey. Let's try another subset of (SName, FName). Its closure includes all other attributes so it is a superkey. It is minimal because we already have observed that no single attribute in this relation is a key. So one of the candidate keys is **(SName, FName)**.

Is there any other candidate key? Let's try (PName, SName, shirtNo). It is not a superkey because its closure does not include date and FName. Let's try (PName, shirtNo, FName). Its closure includes all attributes so it is a superkey. But is it minimal? We find the closure of all its possible subsets (that is, (PName,shirtNo), (PName,FName), (FName,shirtNo), PName, shirtNo, FName) and find that none of its subset is a superkey. Hence **(PName, shirtNo, FName)** is a candidate key.

The relation R8 (AUTOGRAPH) contains two possible candidate keys (SName, FName) and (PName, shirtNo, FName). Any one of these can be selected as primary key. In **AUTOGRAPH** relation, we select (SName, FName) as the primary key.

5 marks for the BCNF decomposition in 5 relations (Team, Player, Stadium, Fan, Autograph).

2 marks for all candidate keys of each table