August 2014 Question 1 [*This question is worth a total of 40 marks.*]

A phone company wants to set up their own App Store for mobile devices. Requirements analysis for the controlling database highlights the following information about what must be recorded.

- Every app in the store needs a unique name, a publisher, and a rating.
- There are two subclasses of app: a *premium* app has to be paid for before installation; a *freemium* app is free to download, but has in-app purchases which cost money.
- The database should record the price of each premium app.
- Each user of the store is identified by their email address.
- A user may have several subaccounts, each identified by a nickname.
- The database needs to record which users have installed which apps.
- Users can use subaccounts to restrict access to freemium apps: the database needs to record which nicknames are allowed to run which ones.

(a) Draw an entity-relationship diagram to represent this information. [*20 marks*]

The app store groups apps into *themes* such as "Games", "News + Magazines", or "Health + Fitness". An app can be in multiple themes, and each theme can have a current "Top App". This is captured by the following SQL data declarations.

```
create table App (
  name      varchar(30),
  publisher varchar(25),
  rating    integer,
  primary key (name)
)

create table Theme (
  title   varchar(20),
  topApp  varchar(30),
  primary key ( title ),
  foreign key (topApp) references App(name)
)

create table InTheme (
  name  varchar(30),
  title varchar(20),
  primary key (name,title),
  foreign key (name) references App,
  foreign key ( title ) references Theme
)
```

(b) What do the terms "arity" and "cardinality" mean when describing database tables? [*2 marks*]

QUESTION CONTINUES ON NEXT PAGE

(c) Write relational algebra expressions to compute the following.

(i) The name of the top app in the "Games" theme.

(ii) For every app in the "Games" theme, its name and rating. [*6 marks*]

(d) Write expressions in the tuple-relational calculus that express the following queries.

(i) The names of all apps in the "Office" theme.

(ii) The publishers of all top apps. [*6 marks*]

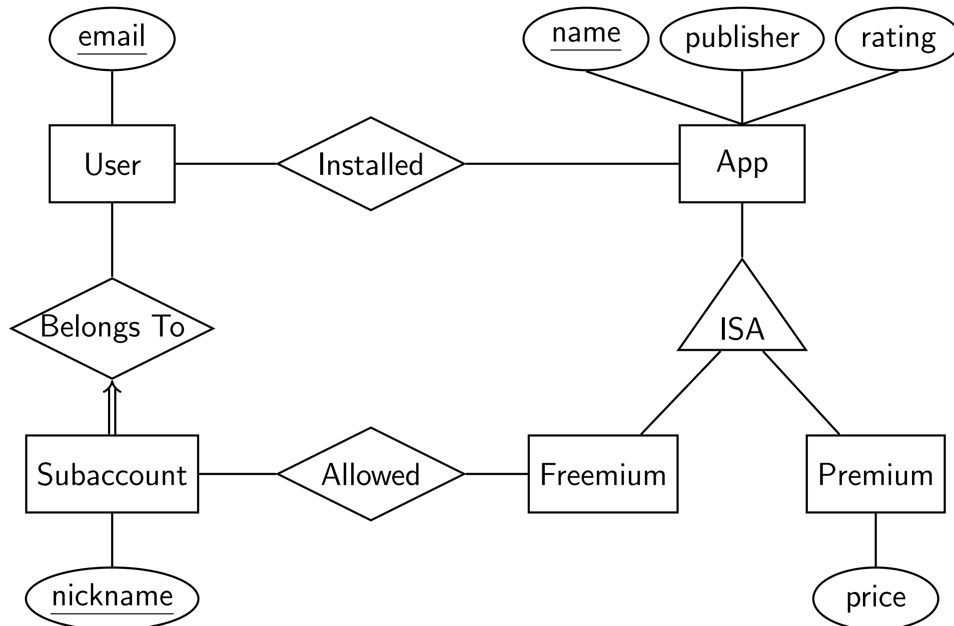
(e) Write SQL queries to answer the following questions.

(i) How many apps are there in the database?

(ii) What is the highest and lowest rating given to apps in the "Utilities" theme? [*6 marks*]

Notes on August 2014 Question 1

(a) The following ER diagram captures the information required.



Notice that the ISA relationship means that Freemium and Premium app entities don't need their own copies of the name, publisher and rating attributes.

The constraints here are as follows.

- *Total participation* of Subaccount in BelongsTo, as every subaccount must belong to some user. Shown by a double (or thick) line in the diagram joining them.
- *Key constraint* between Subaccount and BelongsTo, as every subaccount must belong to at most one user. Shown by an arrowhead on the line in the diagram.

Together these capture that each subaccount belongs to exactly one user.

Some answers made Subaccount a weak entity, with Belongs To as the identifying relationship. This is a reasonable design choice: in particular, if we expect that nickname is not unique across all subaccounts of all users. To show this in the diagram, both Subaccount and BelongsTo should have a double (or thick) outline; and nickname attribute should have a double (or dotted) underline.

Note that it's not enough to just draw a double outline on Subaccount — that entity is involved in two different relationships, Belongs To and Allowed, and we have to show which one is used to uniquely identify instances of the weak entity.

Some students added publisher to the key for the App entity. This is wrong, as the scenario already states that app names are unique; and keys should be minimal.

Some students put an ISA subclass relationship between Subaccount and User. This is incorrect: a subaccount is not a special kind of user; and there can be multiple subaccounts per user. (This is a "has-a" relationship, not an "is-a" relationship.)

Some students incorrectly left out the double line between Subaccount and Belongs To. This is needed because every subaccount must correspond to some user.

Several students added an arrowhead to the line joining App to Installed. That's wrong: such a key constraint would require that each app be installed by at most one user.

Note that the scenario says "the database needs to record which nicknames are allowed to run which" freemium apps. That is why there is a relationship Allowed between Nickname and Freemium in the diagram above. Alternatives like Forbidden, Restricted, or Permission with a boolean yes/no attribute, are close but do not exactly satisfy the scenario given.

(b) The *arity* of a database table is the number of columns (fields, attributes) it has. The *cardinality* of a database table is the number of rows (tuples, records) it contains.

This is bookwork; although with variations following all the different names used to describe the columns and rows of a database table.

(c) The following relational algebra expressions compute the required sets.

(i) $\pi_{\mathsf{topapp}}(\sigma_{\mathsf{title}='\mathsf{Games}'}(\mathsf{Theme}))$

(ii) Either $\pi_{\mathsf{name,rating}}(\sigma_{\mathsf{title}='\mathsf{Games}'}(\mathsf{InTheme}) \bowtie \mathsf{App})$
or $\pi_{\mathsf{name,rating}}(\sigma_{\mathsf{title}='\mathsf{Games}'}(\mathsf{InTheme} \bowtie \mathsf{App}))$ will do.

Note that because InTheme and App share the foreign key name we can use a natural join "$\bowtie$".

Some students explicitly described the join, with "$\bowtie_{\mathsf{Intheme.name=App.name}}$"; that's also correct.

Taking the join Theme $\bowtie$ App is incorrect, as these tables do not share any fields.

(d) The queries given can be captured by these tuple-relational calculus expressions.

(i) $\{ R \mid \exists X \in \mathsf{InTheme} \,.\, X.\mathsf{title} = \mathsf{'Office'} \wedge X.\mathsf{name} = R.\mathsf{name} \}$

(ii) $\{ R \mid \exists T \in \mathsf{Theme}, A \in \mathsf{App} \,.\, T.\mathsf{topApp} = A.\mathsf{name} \wedge A.\mathsf{publisher} = R.\mathsf{publisher} \}$

In both cases an auxiliary relation R is needed to pick out the single desired field.

Note that a tuple-relational calculus expression returns a set of records, not individual fields. Expressions beginning $\{ X.\mathsf{name} \mid \dots \}$ or similar are incorrect, and not meaningful in TRC. Use an auxiliary relation like R instead.

(e) SQL:

(i) **select count**($*$) **from** App
or **select count**(name) **from** App is fine too.
There is no need to add **distinct**, because each name is already unique.

(ii) **select min**(rating), **max**(rating)
from App, InTheme
where App.name = InTheme.name **and** InTheme.title = 'Utilities'

It's possible, but not necessary, to use an explicit join for this.

select min(rating), **max**(rating)
from App **join** InTheme **on** App.name = InTheme.name
where InTheme.title = 'Utilities'

Any SQL query engine should compile both of these to exactly the same execution plan.

Here **count**, **min** and **max** are *aggregate operations* of SQL. These go beyond relational algebra and tuple-relational calculus to compute summary values over the multisets that are returned by basic queries.

May 2015 Question 2 [*This question is worth a total of 40 marks.*]

The following XML document captures some information about a book publisher's catalogue: in this case, the non-existent publisher *Animal Press*.

```
<?xml version="1.0"?>
<!DOCTYPE publisher SYSTEM "publisher.dtd">
<publisher>
    <name>Animal Press</name>
    <imprint>
        <title>Tapir Tomes</title>
        <tag>TT</tag>
        <book code="717899" type="hardback">
            <title>
                Beyond the Trees
            </title>
        </book>
        <book code="833908">
            <title>
                Never Before
            </title>
        </book>
    </imprint>
</publisher>
```

A full catalogue would list many different *imprints* of the publisher — different brand names they use to sell books — with for each imprint a short unique *tag* and a list of many books. Every book has a unique code, and is either *hardback* or *paperback*, with the default being paperback if not specified.

(a) Draw the tree of the *XPath data model* for this XML document [*10 marks*]

(b) Write out a DTD which describes this document and any other similar publisher's catalogue, suitable for the "publisher.dtd" file referenced. [*10 marks*]

(c) Write XPath expressions to obtain the following information from such a document.

   (i) A list of all the imprint tags.

   (ii) The title of the book with code 823095.

   (iii) The title of every imprint that includes at least one hardback book.

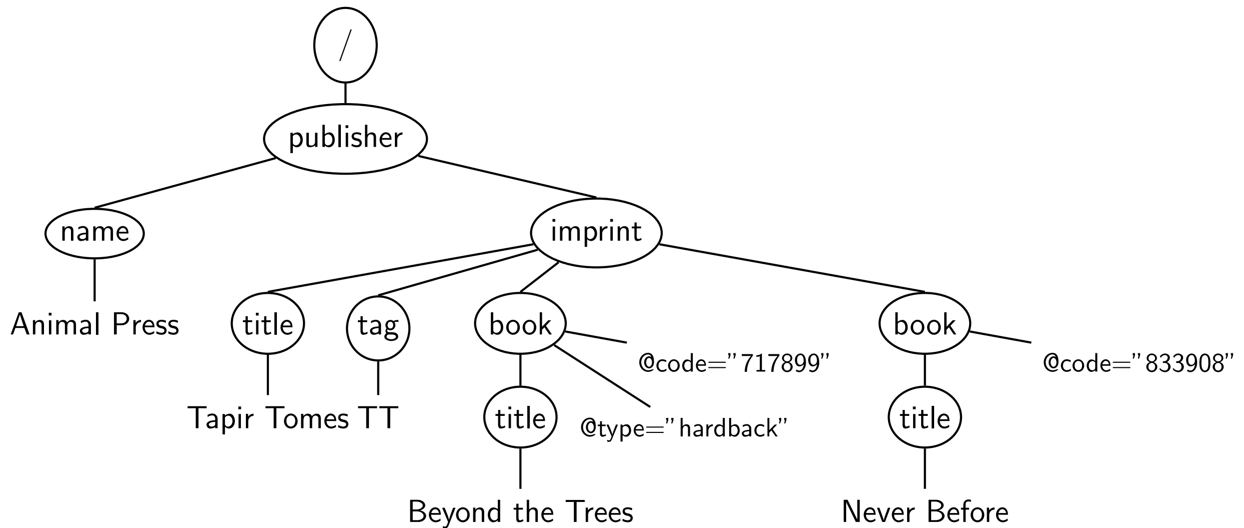
[*6 marks*]

(d) The *Animal Press* themselves keep this information in a relational database with two linked tables: Imprint and Book. Write suitable schemas for these tables in the SQL Data Declaration Language. [*8 marks*]

(e) Based on your schemas, write SQL queries to find out the information required for each item in part (c) above. [*6 marks*]

Notes on May 2015 Question 2

(a) The following tree is the XPath data model of the XML file given.



The left-to-right ordering of element nodes matters: for example, the "name" node must be to the left of the "imprint" node. Ordering of attribute nodes like "@type" and "@code" is arbitrary. The important thing here is not that ordered or unordered is "better", but that you need to be able to work correctly with each as required.

Notice that there is a single root node "/", above publisher. Text in the leaf nodes should not be in quotation marks.

Remember that there is a 1-1 translation between the textual syntax of XML and the tree model of XPath. This means that the tree should not add, remove or change anything that is in the XML text. Several students incorrectly made their own modifications — for example, changing the "title" node to an "imprint−title" node, putting in ". . . " for further nodes, or adding an extra attribute to indicate that "Never Before" is a paperback book. All of these are errors, as the resulting tree does not match the XML text given.

This also happens sometimes with questions posed the other way round: from a tree, give the XML text. Again, it's an error to change the names of nodes or alter the content.

(b) The following is a suitable contents for the "publisher.dtd" file.

```
<!ELEMENT publisher (name,imprint+) >
<!ELEMENT imprint (title,tag,book+) >
<!ELEMENT book (title) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT tag (#PCDATA) >
<!ATTLIST book code CDATA #REQUIRED>
<!ATTLIST book type (hardback|paperback) "paperback" >
```

The ordering of lines doesn't matter: the DTD language is purely declarative, and each individual declaration can refer freely to any other.

Notice that there is only one declaration of the "title" element, even though this is used in two different settings (as the title of an imprint, and of individual books). Repeating the declaration is an error.

There are some legitimate variations on these DTD declarations: for example, using "(name,imprint*)" to indicate that a publisher element might have zero or more imprints rather than one or more. However, using just "(name,imprint)" is an error, as that limits to structure to only one imprint — the question specifically states that a publisher will have many different imprints.

Several students wrote someting like "(title tag—book+)—" for the imprint element. Using a vertical bar "—" here is entirely wrong, as it denotes alternatives (either a title *or* a tag *or* a bool+). The correct connective is the comma "," to denote each item is required in order.

The **ATTLIST** declaration for the type of a book uses an explicit choice to indicate that every book is either hardback or paperback; and that if no value is provided then the default is paperback.

It's possible to put both book attributes into a single **ATTLIST** line:

<!**ATTLIST** book code **CDATA** #**REQUIRED** type (hardback|paperback) "paperback">

(c) (i) Here are three possible variations on an answer:

- //tag/text()
- /publisher/imprint/tag/text()
- //imprint/tag/text()

Note that the final text() test is necessary to pick out the text itself at the leaf of the tree, rather than the element node that is parent to it.

(ii) A couple of possible answers:

- //book[@code="823095"]/title/text()
- //title [../ @code="823095"]/text()

The first of these picks out books with the right code, and then finds the title text; the second gathers titles, and checks the code of their parent book node. Both give the same result.

(iii) Here are a range of possible solutions.

- //imprint[.//@type="hardback"]/title/text()
- //imprint[book/@type="hardback"]/title/text()
  These two pick out imprints containing a book that is a hardback, and then return the imprint title.
- //book[@type="hardback"]/../title/text()
- //book[@type="hardback"]/ancestor::imprint/title/text()
  These two solutions select all hardback books, then navigate up the tree to find out which imprint they come from, and then its title.
- //imprint[count(book[@type="hardback"])>0]/title/text()
  This solution came from a student, and was completely unexpected. I didn't cover the XPath "count()" function in the course, but it's entirely legitimate and, if used correctly, makes for a reasonable answer to the question. Here it appears in a predicate to identify those imprint nodes which contain more than zero book nodes with attribute type of "hardback".

(d) The table declarations below capture the information held in the XML document.

```
create table Imprint (
    title varchar(60) not null,
    tag   varchar(6),
    primary key (tag)
)

create table Book (
    title    varchar(120) not null,
    code     varchar(6),
    type     varchar(10),
    imprint  varchar(6) not null,
    primary key (code),
    foreign key (imprint) references Imprint(tag)
)
```

The exact size of each varchar is not too important. Fields from the primary key are always **not null**, so don't need that declared; however, it's also fine to declare them **not null** explicitly. The foreign key imprint of Book should be **not null**, as each book must belong to some imprint. I've also made the title of both books and imprints **not null**.

I've listed the type of a book as a varchar field, which I expect to take the value "hardback" or "paperback" as in the XML. It would be possible to code this as a boolean, but strictly that's a deviation from the original XML.

Some students renamed the two title fields to book_title and imprint_title. That's not necessary: the ambiguity can always be resolved by referring to them in SQL with the qualified names Book.title and Imprint.title.

Several students made the error of having a book field as a foreign key in the Imprint table. This is entirely wrong: each imprint can contain several books, so there is no single book to point at from the record of a specific imprint. It is sensible to have a foreign key imprint in the Book table, though, so that for each book we can identify the unique imprint to which it belongs.

I've used the short imprint tag as primary key for the Imprint table. It would be possible to use the title field as primary key instead, as these should also be unique. The foreign key in Book would then also be the imprint title, and the solution to the last part of (e) becomes slightly simpler.

(e) The following SQL queries fetch the information required.

(i) **select** tag **from** Imprint

(ii) **select** title **from** Book **where** code = '823095'

(iii) **select distinct** Imprint.title
from Imprint, Book
where Book.imprint=Imprint.tag **and** Book.type='hardback'

Notice the use of **distinct** here to avoid duplication where an imprint contains more than one hardback book. Also, that the **select** statement uses the full-qualified field name Imprint.title to avoid confusion with the Book.title field.

A few students gave an answer similar to this but with the error of missing out either one of the two tables (**from** Imprint, Book) or the crucial test that links them together (Book.imprint=Imprint.tag).

It's possible to use an explicit **join** here instead.

select distinct Imprint.title
from Imprint **join** Book **on** Book.imprint=Imprint.tag
where Book.type='hardback'

This is exactly equivalent to the query given earlier.

If the Imprint table declaration uses title as its primary key rather than tag, this question becomes a little simpler as each Book record will already include the complete imprint title as a foreign key field.

select distinct Book.imprint
from Book **where** Book.type='hardback'

It's possible to argue about whether there might be efficiency problems in using a lengthy text field like the imprint title as a primary key. On the whole, though, these discussions are only sensible in the context of a specific, known, database engine; which may well have perfectly good handling of such key fields through hashes or other methods.