Assignment 3

MyMyUNSW Course Advisor

Last updated: **Wednesday 23rd October 11:10pm**
Most recent changes are shown in red ... older changes are shown in brown.

[Assignment Spec] [Database] [Rules] [Schema] [Test Cases] [Fixes+Updates] [Extras]

Downloads: ass3.zip, ass3.tgz, ass3.db.bz2, ass3.db.zip
Note that `ass3.tgz` and `ass3.zip` contain the same material.
Each archive contains supplied PHP code plus `updates.sql`.
Similarly, `ass3.db.bz2` and `ass3.db.zip` contain the same database dump.

Aims

This assignment aims to give you practice in

- further use of SQL and PLpgSQL
- writing scripts in PHP that interact with a database

The goal is to complete the functionality of some command-line tools via a combination of database code and PHP code.

Optionally, you can install a web interface to some of the operations and incorporate your PHP code into this interface.

Summary

Submission: Login to Course Web Site > Assignments > Assignment 3 > [Submit] > upload required files

Required Files: ass3.php, updates.sql

Deadline: Wednesday 30 October 23:59

Late Penalty: 0.12 *marks* off the ceiling mark for each hour late

This assignment contributes **15 marks** toward your total mark for this course.

How to do this assignment:

- read this specification carefully and completely
- create a directory for this assignment
- unpack the supplied files into this directory
- login to `grieg` and run your PostgreSQL server
- remove your Assignment 2 database
- set up a copy of the supplied database (called e.g. `ass3`)
- edit the configuration parameters for the PHP code (in `lib/defs.php`)
- re-acquaint yourself with the database
- familiarise yourself with the PHP code base
- complete the tasks below by editing the files `lib/ass3.php` and `updates.sql`
- submit these files via WebCMS (you can submit multiple times)

Details of the above steps are given below. Note that you can put the files wherever you like; they do not have to be under your `/srvr` directory. You also edit the PHP and SQL files on hosts other than `grieg`. The only time that you need to use `grieg` is to manipulate your database or to run your PHP scripts.

You will probably not be able to fit both the database for Assignment 2 and the database for Assignment 3 into your PostgreSQL server on `grieg` (because of your quota on `/srvr`). The Assignment 3 database is substantially different to the Assignment 2 database, so you should first remove your database from Assignment 2 before creating the Assignment 3 database.

Introduction

In Assignment 2, we introduced the MyMyUNSW schema and database for managing UNSW academic information. In this assignment, we will be working with almost the same schema, but with some new tables and some additional data. The differences between the Assignment 2 and Assignment 3 databases:

Trimmed `Programs`, `Streams`, `Subjects` tables

The `firstoffer` and `lastoffer` fields in these tables were supposed to indicate a range of semesters during which a particular version of the academic object was offered. The values for these fields were estimated based on enrolment data, and were flakey. Also, since we don't have rules data for old program and stream versions, maintaining just the meta-data seemed like a waste of time. The database now maintains just a single version of each academic object (program, stream or subject), and so the `firstoffer` and `lastoffer` fields have been removed from all three tables. A downside of this is that students enrolled many years ago will need to be assessed against the current version of their program/stream. This will probably mean that they are no longer eligible to graduate (because they haven't completed the current core courses). Oh well ...

In the `Subjects` table, the fields that were intended to hold the textual versions of the `_excluded`, `_equivalent`, and `_prereq` subjects have been removed. The `Subjects.excluded` field is a reference to an academic object group indicating all of the subjects that may not be taken once the current subject is completed. The `Subjects.equivalent` field is a reference to an academic object group for subjects that are deemed to be equivalent. Note that the current subject appears in this list, which is referenced from all of the other subjects in the "equivalence group". A student may not complete more than one subject from an equivalence group. Each subject is linked to its pre-requisites via the new `Subject_prereqs`

table (see below).

Added `Rules` table

The `Rules` table holds information associated with the requirements for completion of degrees and subject pre-requisites. The data does in this table does not always fit with what's in the UNSW Handbook, but is the best we could do from the available data resources. If a rule in the database conflicts with what you know in reality (e.g. with what you find in the UNSW online handbook) don't worry about it; you should treat the *database* as the *new reality*.

Subjects are linked to their pre-requsities via the `Subject-prereqs` table, which implements an n:m relationship between subjects and rules. If a subject has multiple pre-requisite rules, then they all need to be satisfied before a student is eligible to take the subject. Pre-requisite rules typically mention subjects (or groups of subjects) which need to be completed, but may also include requirements to achieve a certain WAM or to have passed a certain number of UOC.

The pre-req data was obtained by crawling the UNSW Handbook. Since the pre-reqs in the handbook are written in plain english and don't follow a standard format, not all pre-reqs could be converted into database format. If a subject appears in the database without any pre-req data, assume that it has no pre-reqs, even if this is different from reality. If the pre-reqs for some subject are different to what you know, treat the database as correct.

The program and stream requirements data was taken from the MAPPS database. Not all programs and streams have associated requirements in that database. If a program or stream has no requirements, we'll ignore it for the purposes of this assignment.

Note that the rules make extensive used of the `acad_object_groups` table that was considered in Assignment 2.

Added some utility views and functions

You can find out what these functions and views are by examining the database in `psql`. The command `\dv` gives you a list of views, while the following command will give you the definition of one particular view:

```
ass3=# \d+ ViewName
```

Similarly, `\df` gives you a list of functions, and the following command will give you the function definition:

```
ass3=# \df+ FunctionName
```

Note that some functions are overloaded. There are two `transcript` functions: `transcript(int)` gives you transcript records for a given student, regardless of when they studied (based on their UNSW student id); `transcript(int,int)` gives you transcript records for a given student relative to a given semester (based on `People.id` and `Semesters.id` values). The second `transcript` function includes any subjects being studied in the specified semester, but shows them as not yet completed. Use the above `\df+` command to view the definitions for both versions of the `transcript` function.

Aims of this Assignment

As we noted in lectures, the existing MyUNSW system (Campus Solutions) does not currently support two operations that would be extremely helpful to students in managing their enrolment:

- finding out how far they have progressed through their degree program, and what remains to be completed
- checking what are their enrolment options for next semester (e.g. get a list of "suggested" courses)

The main aim of this assignment is to attempt to overcome these problems by implementing the above functionality.

To get a feel for what's ultimately required, try manually doing the following:

a. take a copy of your current academic transcript
b. get a copy of the UNSW Handbook page for your degree in the year you commenced
   (you might also need to go the pages describing plans for your degree to get the full details)
c. try to work out what else you need to do in order to graduate
d. make a list of possible subjects that you might enrol in next semester

Assumptions

The actual task of determining progress and graduation status is rather complicated, and typically requires manual intervention to "allocate" courses to appropriate requirements to maximise the student's position. In order to keep it manageable, we make the following simplifying assumptions:

- There are no enrolment variations, despite the presence of the `Variations` table and despite the existence of courses with a grade of `'T'` indicating credit. You can completely ignore any course with a `'T'`; treat such courses as if they are not in the transcript at all.
- We do not consider double degrees (now called "dual-award programs"). They have complex rules for double-counting various classes of subjects. In addition, few double-degrees have rules in this database.
- Treat the database literally, and interpret everything relative to the rules in the database, even if you know that they don't correspond to reality. As noted above, it was close to impossible to translate all of the rules completely and correctly from the various data sources.

Doing this Assignment

The following sections describe how to carry out this assignment. Some of the instructions must be followed exactly; others require you to exercise some discretion. The instructions are targetted at people doing the assignment on Grieg. If you plan to work on this assignment at home on your own computer, you'll need to adapt the instructions to "local conditions".

If you're doing your assignment on the CSE machines, some commands must be carried out on `grieg`, while others can (and probably should) be done on a CSE machine other than `grieg`. In the examples below, we'll use `grieg$` to indicate that the comand must be done on `grieg` and `cse$`

to indicate that it can be done elsewhere.

Setting Up

The first step in setting up this assignment is to set up a directory to hold your files for this assignment.

```
cse$ mkdir /my/dir/for/ass3
cse$ cd /my/dir/for/ass3
cse$ tar xfz /home/cs3311/public_html/13s2/assignments/3/ass3.tgz
cse$ ls
advisor  codes  lib  studes
ass3.db.bz2 hbook  members  ts
cansat  ingroup  progress updates.sql
cse$ ls lib
ass3.php db.php  defs.php rules.php
```

It would be worth taking a look at the various PHP scripts in this directory and in the `lib/`subdirectory. The files marked in green are the ones you need to modify and submit. You will also need to modify `lib/defs.php` as follows:

```
define("BASE_DIR","/my/dir/for/ass3");
```

If you decide to use a different database name than `ass3`, you will also need to modify the `DB_CONNECTION` value.

The next step is to set up your database:

```
... do priv srvr and source env as usual ...
grieg$ dropdb ass2
grieg$ createdb ass3
grieg$ psql ass3 -f /home/cs3311/public_html/13s2/assignments/3/ass3.db
grieg$ psql ass3
... examine the database contents ...
```

Once you've done the above, the php scripts should be able to connect to your database. You can check this by running the completed script that generates a transcript:

```
# shows all subjects studied up to and including 10s1
grieg$ /srvr/cs3311psql/lib/php535/bin/php ts 3211807 10s1
Gerard Rettke (3211807)

Program/Stream enrolments:
--------------------------
Term  Prog  Stream
09s1  3978  Unknown
09s2  3978  Unknown
10s1  3978  COMPA1

Course enrolments (transcript):
-------------------------------
  Course Term Title                          Mark Grade   UOC
COMP1917 09s1 Computing 1                      57    PS     6
ENGG1000 09s1 Engineering Design               71    CR     6
MATH1081 09s1 Discrete Mathematics             47    PC     6
MATH1131 09s1 Mathematics 1A                   66    CR     6
ACCT1501 09s2 Accounting & Financial Mgt 1A    51    PS     6
COMP1927 09s2 Computing 2                      37    FL     0
ECON1101 09s2 Microeconomics 1                 37    FL     0
MATH1231 09s2 Mathematics 1B                   61    PS     6
ECON1101 10s1 Microeconomics 1                              6
              Overall WAM                      53          36

# shows all subjects studied up to and including 13s2
# since the student stopped in 12s2, that's all we see
grieg$ /srvr/cs3311psql/lib/php535/bin/php ts 3211807 13s2
Gerard Rettke (3211807)

Program/Stream enrolments:
--------------------------
Term  Prog  Stream
09s1  3978  Unknown
09s2  3978  Unknown
10s1  3978  COMPA1
10s2  3978  COMPA1
11s2  3978  COMPA1
12s1  3978  COMPA1
12s2  3978  COMPA1
13s1  3978  COMPA1

Course enrolments (transcript):
```

```
------------------------------
  Course Term Title                          Mark Grade   UOC
COMP1917 09s1 Computing 1                      57    PS     6
ENGG1000 09s1 Engineering Design               71    CR     6
MATH1081 09s1 Discrete Mathematics             47    PC     6
MATH1131 09s1 Mathematics 1A                   66    CR     6
ACCT1501 09s2 Accounting & Financial Mgt 1A    51    PS     6
COMP1927 09s2 Computing 2                      37    FL     0
ECON1101 09s2 Microeconomics 1                 37    FL     0
MATH1231 09s2 Mathematics 1B                   61    PS     6
ECON1101 10s1 Microeconomics 1                 70    CR     6
COMP1927 10s2 Computing 2                      27    FL     0
ECON1102 10s2 Macroeconomics 1                 72    CR     6
INFS1602 10s2 Info Systems in Business         72    CR     6
INFS1603 10s2 Business Databases               67    CR     6
COMP1927 11s2 Computing 2                      38    FL     0
INFS2603 11s2 Business Systems Analysis        70    CR     6
MARK1012 11s2 Marketing Fundamentals           62    PS     6
COMP1927 12s1 Computing 2                      71    CR     6
COMP2121 12s1 Microprocessors & Interfacing    71    CR     6
INFS2607 12s1 Networking and Infrastructure    59    PS     6
COMP2041 12s2 Software Construction            57    PS     6
              Overall WAM                      58          96
```

If this doesn't work, you'll probably get some error messages to give you a clue as to what's wrong. Generally, fixing your `lib/defs.php` confiugration, or running the commands on the correct server, etc. will fix the problem.

It will probably get a bit tedious typing the full name of the PHP interpreter each time. In fact, if you're on `grieg` and you've source'd your `/srvr/`*`YOU`*`/env` file, it should be using the correct PHP interpreter if you simply type:

```
grieg$ php ts 3211807 13s2
```

If this produces an error suggesting that `pg_connect` is unknown, then you're using the wrong PHP interpreter. CSE's standard PHP installation on Grieg does not include the PostgreSQL library, and cannot be used for this assignment. You can check that you're using the correct PHP by:

```
grieg$ which php
/usr/X11/bin/php  ... this is the WRONG interpreter ...
grieg$ source /srvr/YOU/env
grieg$ which php
/srvr/cs3311psql/lib/php535/bin/php
```

Once you can look at transcripts, you're set up and ready to start editing `lib/ass3.php` and `updates.sql`.

> How can you find other interesting transcripts to look at? The answer is to think of some properties of a transcript that might make it interesting, and then ask a query to get information about any students who have these kinds of transcripts. I used the following query to find some students (including the one in the example). Work out what it does and then try variations to find other kinds of interesting students:
>
> ```
> select p.unswid,pr.code,termName(min(pe.semester)),count(*)
> from   People p
>          join Program_enrolments pe on (pe.student=p.id)
>          join Programs pr on (pe.program=pr.id)
> where  pr.code like '3%'
> group  by p.unswid,pr.code
> having count(*) > 5;
> ```
>
> In fact, we've made your life a bit simpler by supplying a PHP script called `studes` which gives similar information to the above query, for all students in the database. You can use it in conjunction with the Unix `grep` command to find interesting students. Or you can run queries like the above ...

Note that the supplied PHP code has a significant set of functionalities already included, and it would be useful for you to explore what it currently does and then read the code and examine the database to find out how it does it. This is especially true of `lib/rules.php` which supplies the `showRule()` and `ruleName()` functions that are useful/required in various tasks.

Submission and Testing

We will test your submission as follows:

- create a testing subdirectory
- untar the `ass3.tgz` file into that directory
- load your submitted `ass3.php` and `updates.sql` files over the top of the standard ones
- create a new database *`TestingDB`* and initialise it with `ass3.db`
- run the command: `psql` *`TestingDB`* `-f updates.sql` (using your `updates.sql`)
- manually run a series of tests using our `members`, `ingroup`, `cansat`, `progress` and `advisor` scripts

- manually inspect your submitted PHP code and SQL code

Your submitted code must be *complete* so that when we do the above, your PHP will work just as it did in your assignment directory and with a database that looks identical to yours (which is what your `updates.sql` script is supposed to ensure). If your code does not work when installed for testing as described above and the reason for the failure is that your `updates.sql` did not contain all of the required definitions, or you did not submit all of the required PHP files, you will be penalised by a 3 mark administrative penalty.

Some things that you can do wrong:

- make changes to files *other than* `ass3.php` and `updates.sql`
- forget to include all of your PHP functions in `ass3.php`
- forget to include all of your SQL and PLpgSQL definitions in `updates.sql`

Before you submit, it would be useful to test out whether the files you submit will work by following a similar sequence of steps to those noted above.

Tasks

Use the MessageBoard to clarify the specification. We will also (eventually) make some sample output available for you to compare against your output for various test cases.

Each task has available a PHP script which invokes the function you are required to implement and displays the result of the function in a standard format (which I can use for auto-marking). The sample output will assume that you are using these scripts to test your functions. Of course, you are free to modify them e.g. to add debugging `print_r()` calls, but eventually they will be tested as supplied.

In implementing your functions, you are free to partition the functionality however you like between the database and the PHP scripts. In the past, some students have solved similar assignments to this by writing just about everything in SQL views and PLpgSQL functions and using PHP simply as a vehicle for collecting the results. Others have done most of the computational work in PHP. Do whatever you feel most comfortable with, but make sure that you limit your changes just to the PHP scripts that you are required to submit.

Task A: Object Group Members (2 marks)

The `members` script lists the members of an academic object group, by printing their codes (e.g. `COMP3311`), one per line. It has the following command-line arguments:

```
grieg$ php members GroupID
```

where *GroupID* is an internal ID (`Acad_object_groups.id`) identifying one academic object group.

Some examples of use (more examples can be found on the [Sample Test Cases] page):

```
grieg$ php members 5672
ENVS4101
ENVS4102
ENVS4103
ENVS4104
```

The `members` script makes use of the function

```
function membersOf($db, $groupID)
```

where the parameters are:

- `$db` ... an open database connection
- `$groupID` ... the `Acad_object_groups.id` value identifying the group

You should complete the definition of this function in `lib/ass3.php`. Any SQL or PLpgSQL functions that you write to support it should be placed in the `updates.sql` file.

This function returns a pair of values as a PHP array. The first element of the pair is the type of objects in the group. The second element is an array of codes of the group members (in alphabetical order). For the above example, the function would return:

```
array("subject",array("ENVS4101","ENVS4102","ENVS4103","ENVS4104"))
```

A reminder: each group contains a collection of objects all of the same type, which can be `program`, `stream` or `subject`. A `program` group would return codes like `3978` or `1650`; a `stream` group would return codes like `COMPA1` or `ENGLB1`; a `subject` group would return codes like `COMP3311` or `MATH1131`.

Groups can be defined either by enumeration or via a pattern or a query. You dealt with enumerated groups and simple patterns in Assignment 2. Now you must deal with all of the patterns, including the following special cases:

- "*abc[XYZ]def*" means anything matching either "*abcXdef*" or "*abcYdef*" or "*abcZdef*"
- "{*Code*$_1$;*Code*$_2$;...}" is equivalent to "*Code*$_1$, *Code*$_2$, ..." (at least for membership purposes)
- "!*Pattern*" excludes any object matching the pattern
- "*Pattern*/F=*FacId*" includes only matching courses objects offered under the specified faculty

Some groups of subjects are very large, and not worth expanding in full (at least for our purposes), so we treat them specially. Any subject group that is defined by a pattern starting with one of following strings should simply return the pattern itself as the code (i.e. do not expand it):

- `"GENG"`, `"GEN#"`, `"FREE"`, `"####"`, `"all"`, or `"ALL"`

The above should be done when either the pattern is stand-alone or if it appears as one of a comma-separated list of patterns e.g. `COMP3###,GENG####"`. In the example case, the function should return a list of all `COMP3` course codes, followed by a single `"GENG###"` entry.

Note the intpretations of the following special patterns (not useful for the `members()` function, but relevant later):

- `"GENG####"` means any Gen Ed course (code == "GEN....") offered outside the "home" faculty
  (note that the "home" faculty cannot be determined without considering program/stream enrolment as in Task C, so we ignore it here)
- `"FREE####"` or `"FREE"` means any non-Gen Ed course (code != "GEN.....")

Note that some enumerated groups include sub-groups that are defined by having additional `acad_object_groups` that refer to the main group via the `parent` attribute. This is essentially the equivalent to the `{...;...}` patterns in the pattern-based groups.

You can assume that there will be no negated `acad_object_groups` (i.e. that there is no tuple where the `acad_object_groups.negated` attribute is true).

If an academic object group is defined using a query, you can assume that the query returns tuples containing the `id` and `code` of each object i.e. that any embedded query is of the form:

```
select id,code from ...
```

Task B: Membership Check (3 marks)

The `ingroup` script determines whether a particular academic object is a member of a specified group. It has the following command-line arguments:

```
grieg$ php ingroup Code GroupID
```

where *Code* is the identifying code for one academic object (either a `Subjects.code` or `Streams.code` or `Programs.code` value) and *GroupID* is an internal ID (`Acad_object_groups.id`) identifying one academic object group

Some examples of use (more examples can be found on the [Sample Test Cases] page):

```
grieg$ php ingroup COMP1917 99999
Invalid AcObjGroupID (99999)
# Group 6826 is a subject group defined as COMP3###,COMP4###,COMP9###
grieg$ php ingroup COMP3311 6826
yes
grieg$ php ingroup SENG4921 6826
no
# Group 4070 is a subject group for Free Electives
# In theory, COMP3311 could be counted as a free elective
grieg$ php ingroup COMP3311 4070
yes
```

The `ingroup` script makes use of the function

```
function inGroup($db, $code, $groupID)
```

where the parameters are:

- `$db` ... an open database connection
- `$code` ... a `Subjects.code`, `Streams.code` or `Programs.code` value
- `$groupID` ... the `Acad_object_groups.id` value identifying the group

You should complete the definition of this function in `lib/ass3.php`. Any SQL or PLpgSQL functions that you write to support it should be placed in the `updates.sql` file.

This function takes an object code and determines whether this object is one of the objects identified via the group's definition.

For this, we cannot simply rely on the `members()` function from Task A, because it abbreviates some groups (e.g. groups defined by patterns containing elements like `FREE####`). If the group is a set of specific objects or a pattern that does not produce too many results (e.g. `"COMP3###"`), the checking is easy ... just expand the set of objects in the group and check whether the specified object is a member of that set. For matching against the patterns with very many matches (i.e. the kinds of patterns which were abbreviated in Task A), use the following guidelines:

- `"FREE"`, `"####"`, `"all"`, or `"ALL"` include any subject whose first three characters are not `"GEN"`
- `"GENG"`, `"GEN#"` include any subject whose first three characters are `"GEN"`
- `"GENG####/F=SCI"` includes any subject whose first three characters are `"GEN"` and which is offered by the Science Faculty or some school under the Science Faculty
- `"GENE####"` includes any subject whose first four letters are `"GENE"`, etc.
- `"!Pattern"` excludes any subjects matching the pattern from the group

Note that there are also groups of streams and programs in the database, and these must be handled appropriately.

Task C: Satisfaction Check (2 marks)

The `canSat` script determines whether a student could use a given academic object towards satisfying a given rule. It has the following command-line arguments:

```
grieg$ php cansat Code RuleID StudentID Semester
```

where *Code* is the identifying code for one academic object (either a `Subjects.code` or `Streams.code`) *RuleID* is an internal ID (`Rules.id`) identifying one rule, *StudentID* is UNSW student ID (i.e. `People.unswid`) and *Semester* is a four character code identifying a semester (e.g. `"12s2"`). (Note that program codes are not relevant here since there are no rules that require certain programs to be completed as part of a requirement.)

The semester is required in order to work out which program and stream(s), the student is enrolled in. For most cases, the student's current enrolment is not relevant, but it becomes important in determining whether a Gen Ed subject can satisfy a Gen Ed rule. Students can only count Gen Ed courses taken from outside their "home" faculty, where the home faculty is determined from the organizational units who offer the program/stream(s).

Some examples of use (more examples can be found on the [Sample Test Cases] page):

```
# Rule 10389 specifies COMP[349]####, part of the student's 3978 program
$ php cansat COMP3311 10389 3225416 10s1
yes
$ php cansat SENG4921 10389 3225416 10s1
no
# Rule 11461 specifies Gen Ed, part of the student's 3648 program
$ php cansat GENE8001 11461 3214627 10s2
no  # GENE8001 *is* a Gen Ed, but it's offered under the Engineering faculty
$ php cansat GENT0410 11461 3214627 10s2
yes  # GENT0410 is a Gen Ed subject from the Arts Faculty
$ php cansat SENG2010 11461 3214627 10s2
no  # SENG2010 is not a Gen Ed subject
```

At this stage, you do not have to worry about whether the student has already completed the subject. Similarly, you can ignore any maturity rules that might prevent enrolment in the subject in the given semester. The satisfaction check is relatively straightforward: a subject/stream can potentially satisfy a rule if it is one of the subjects/streams in the academic object group associated with the rule. The one exception to this relates to Gen Ed subjects, as noted above.

The `cansat` script makes use of the PHP function

```
function canSatisfy($db, $code, $ruleID, $enrolment)
```

where the parameters are:

- `$db` ... an open database connection
- `$code` ... a `Subjects.code`, `Streams.code` or `Programs.code` value
- `$ruleID` ... the `Rules.id` value identifying the rule
- `$enrolment` ... an array describing the student's enrolment status (see below)

You should complete the definition of this function in `lib/ass3.php`. Any SQL or PLpgSQL functions that you write to support it should be placed in the `updates.sql` file.

The `$enrolment` parameter is an array containing the `Programs.id` of the program in which the student is enrolled during the given semester, as well as an array of `Streams.id` values for all of the streams that they are enrolled in for that semester. For example, the student with ID 3211807 used in the `ts` examples above, was enrolled in the stream `COMPA1` in the program `3978` in semester 10s2. His `$enrolment` value would be:

```
array(554, array(298))
```

where 554 is the ID value for program `3978` and 298 is the ID value for stream `COMPA1`. If we considered this same student's enrolment in 09s1, where his stream was unknown (undeclared), then the `$enrolment` value would be:

```
array(554, array())
```

In this case there is no useful stream value to use in determining a faculty for Gen Ed rule checking; simply use the faculty for the program.

Many students take two streams concurrently under one program. An example of such a student is one with ID 3351736, who is enrolled in streams `COMPA1` and `COMPN1` (ID=321) under the program 3978. His `$enrolment` value would be:

```
array(554, array(298, 321))
```

The `canSatisfy()` function needs to take account of all of the faculties related to the program and stream(s) involved. In the above example, all of them are under the Engineering Faculty, but this would not necessarily be the case in a combined degree (e.g. Commerce/Engineering, where one stream would come from Engineering and the other from Commerce) or even in some single-degrees (e.g. Arts, where you could take a History major under Arts and a Computer Science minor under Engineering).

Note that the `$enrolment` value is computed for you by the `cansat` script, so you don't need to do it yourself. However, you will need to *use* this value in implementing your `canSatisfy()` function.

The above discussion assumes that the rule involved is an RQ rule with an associated object group, a DS rule with an associated stream group, or a CC, PE, FE or GE rule, with an associated subject group. If such a rule has no associated group (`ao_group is null`) or if the associated group is empty, assume that the rule cannot be satisfied and return false. Also, if the academic object isn't of the same type as the objects in the group, return false. Since the other rule types (e.g. LR, MR, WM, IR) cannot be satisfied on the basis of one academic object, return false for all of these kinds of rules.

Task D: Program Progress (4 marks)

A "progress so far" script (called `progress`) aims to show students a transcript-based view of their progress through their degree program. This script takes two command-line arguments:

```
grieg$ php progress UNSWstudentID SemesterCode
```

Some examples of use (more examples can be found on the [Sample Test Cases] page):

```
grieg$ php progress 3211807 09s2
... show Gerard Rettke's transcript up to and including 09s2 ...
... this would include courses completed in 09s1 and being undertaken in 09s2 ...
grieg$ php progress 3211807 11s1
... show Gerard Rettke's transcript up to and including 11s1 ...
... this would include all courses from 09s1 to 10s2 (he is not enrolled in 11s1) ...
grieg$ php progress 1234567 09s1
Invalid student (1234567)
grieg$ php progress 3211807 20s1
Invalid term (20s1)
... the database only has semesters up to 2015 ...
```

The script displays a "virtual transcript" of the student's progress up to the specified semester. For courses from previous semesters, use the grade/mark values stored in the `CourseEnrolments` table. Treat courses currently being studied (in the specified term, and hence with a null mark and grade) as incomplete.

The semester *S* supplied as the third parameter to `progress` defines the "current" semester. Normally, *S* would be a semester during which the student was enrolled and *S* would be used to determine the student's program/stream enrolment. Also, all subjects enrolled during *S* would be treated as not-yet-completed. Note that the "current" semester could be a summer or winter semester.

If the semester *S* falls after all known program enrolments for this student, then use the last semester *L* of their enrolment to determine which program/stream they were enrolled in, and treat all of their courses as having been completed. If *S* falls before all known program enrolments for this student, then treat the first semester *F* of their enrolment as "current" and do everything relative to *F*. If the student is enrolled before and after the semester *S*, but is *not* enrolled during *S*, then use the most recent semester *R* before *S* where they *were* enrolled to determine the program/streams, and treat all subjects enrolled during *R* as completed.

The `progress` script gets its results by invoking a function (`progress()`) that returns an array of transcript items. This array has two parts: the first part contains transcript information for completed subjects (regular transcript information) while the second part contains a list of TODO items. Each regular transcript item is itself an array containing:

- subject code (e.g. `COMP3311`)
- term code (e.g. `07s2`)
- subject title
- mark (integer in range 0..100, and may be null)
- grade (e.g. `FL`, `HD`, and may be null)
- UOC (UOC awarded for this course; null if failed)
- requirement (which requirement this course was used for)

In terms of PHP data structures, it is an array containing, e.g.

```
array("COMP3311","10s1","Database Systems",75,"DN",6,"Level 3/4 Electives")
array("COMP3231","10s1","Operating Systems",35,"FL",null,"Failed. Does not count")
array("ANAT1006","11s1","Anatomy 1",65,"CR",null,"Fits no requirement. Does not count")
array("COMP1111","11s2","MATS1002",77,"DN",null,"Does not count. Limit exceeded")
... used when a Limit Rule (type=LR) is violated
array("COMP4001","12s1","OO Design",null,null,null,"Incomplete. Does not yet count")
... the above is a current course ...
```

The information in the first six of these fields can be obtained by using the supplied `transcript(int,int)` function. There are three possible cases for the requirement element in the first part of the virtual transcript array:

- if they passed the course, and if the course can be fitted to one the program/stream requirements, then set the value of the element to `ruleName($rid)` (where `$rid` is a `Rules.id` value and the `ruleName()` function is defined in `lib/rules.php`); they should also be awarded the appropriate UOC for the course
- if they are currently studying the course, then set the value of the element to the string `"Incomplete. Does not yet count"`
- if they passed the course, but if the course *cannot* be fitted to one the program/stream requirements, then set the value of the element to the string `"Fits no requirement. Does not count"`
- if they failed the course (not a passing grade), then set the value of the element to the string `"Failed. Does not count"`

The first part of the virtual transcript array ends with an item (array) containing just three elements:

```
array("Overall WAM", WAMvalue, TotalUOCpassed)
```

Note that PHP is fine with having different kinds of elements mixed in an array.

The second part of the virtual transcript array is made up of items, one for each of the incomplete requirements. Order these items first by the rule priority given below (CC then PE then FE then GE then LR). Within each priority group, order by `Rules.id`. Each item is an array with two elements:

- an indication of how much of the requirement is incomplete
- a description of the requirement

For example:

```
array("36 UOC so far; need 9 UOC more", "Free Electives")
array("0 UOC so far; need 6 UOC more", " Level 3 Electives")
```

In order to determine what to put in the first element of each item, you will need to keep track of how many UOC from each requirement has been completed. If the requirement need several courses to be taken, then compute the total UOC for the requirement (use `Rules.min`) and the number of UOC awarded so far, and produce a string like the one above:

```
"UOCdone UOC so far; need UOCleft UOC more"
```

As long as you return an array conforming to the above, the `progress` script will take care of displaying it.

The function used by `progress` to produce the virtual transcript array is defined as:

```
function progress($db, $studentID, $termID)
```

where the parameters are:

- `$db` ... an open database connection
- `$studentID` ... the `People.id` value for the student (*not* their UNSWid)
- `$termID` ... the `Terms.id` value for the specified term

You should complete the definition of this function in the `ass3.php` file. Any SQL or PLpgSQL functions that you write to support it should be placed in the `updates.sql` file.

In order to work out requirements, you should use rules from the program and stream(s) that are associated with the student's "latest" `Program_enrolments` record. If they are enrolled in the "current" semester, then use the program and stream enrolments for that semester; otherwise, use the program and stream enrolments in the most recent semester when they were enrolled.

The rule types that determine progress towards the requirements of a degree are:

- `'CC'` ... how many UOC of your core courses have you completed?
- `'PE'` ... how many UOC of your program electives have you completed?
- `'FE'` ... how many UOC of your free electives have you completed?
- `'GE'` ... how many UOC of your Gen Ed courses have you completed?
- `'LR'` ... have you exceeded the limit on some group of courses (e.g. level 1)?

`'LR'` rules with a `max` value are relevant because they prohibit you from counting courses that might otherwise be used to satisfy an `'FE'` or `'GE'` requirement. `'LR'` rules with a `min` value also need to have UOC allocated to them for each completed relevant subject. `'DS'` rules are relevant to degree progress, but only if you want to determine graduation status; since Task D is primarily focussed on how far you have progressed with respect to courses, you can ignore `'DS'` rules for this question. `'IR'` and `'RC'` rules are advisory only, and can be ignored. `'MR'`, `'RQ'` and `'WM'` rules are used for determining whether you are allowed to take courses, not whether courses contribute to progress, and can also be ignored. Finally, note that there are no advanced standing or exemption variations in this database, so you do not need to handle them.

Hint: use the `transcript(int,int)` function to find out what students have completed; get a combined list of requirements from the program and the stream(s); allocate completed subjects to requirements. Ideally, there should be no duplication of rules between the program and the stream(s); you may assume this, even if there are rogue data items for which this is not true.

Note that the process of allocating courses to rules is not deterministic. There are bound to be cases where one course could be used to satisfy several rules. In real life, the strategy that's adopted is to allocate things in such a way as to maximise the student's progress towards the degree. In a computational setting, this would require determining multiple possible allocations of courses to rules and then choosing the best. This starts to get overly complicated, so you should use the following simple heuristics that tend to will give a reasonable allocation, but maybe not the "optimal" allocation:

- give priority allocating a course to CC rules, then PE, then FE
- GE rules should only be filled with "GEN#####" courses
- if a course could be allocated to two rules at the same priority level, pick the rule with the lowest internal rule ID (simply for the sake of everyone getting consistent output for auto-marking)

And, of course, if a course cannot be allocated to any rule using the above, it should be flagged as not being counted towards the degree.

The GE allocation rule is going to prevent some students from counting some non-GEN##### courses towards their degree which would in reality have been counted as Gen Ed. Don't add more complexity to try to solve this.

Note that it is remotely possible that the group of subjects associated with a CC or PE or FE rule is empty, meaning that the requirement could never be satisfied if it had min>0. If any such cases arise, treat the corresponding rule as if it has been satisfied.

Hint: the `Rules` table and `Acad_object_groups` tables contain the critical data for this question. However, not all attributes in these tables are relevant for solving the problem. In particular, the `glogic` field does not help in dealing with subject/stream groups associated with Rules; to determine how to interpret such groups, use the `min`/max values in the `Rules` table to determine how many subjects in the group need to be taken to satisfy the rule.

Task E: Academic Advising (4 marks)

A script called `advisor` aims to suggest what subjects a student can take in the semester *following* the specified semester/term. (Note that we ignore summer and winter terms for this exercise, so that e.g. 12s1 follows 11s2 and 11s2 follows 11s1). This script takes two command-line arguments:

```
grieg$ php advisor UNSWstudentID Term
```

Some examples of use (more examples can be found on the [Sample Test Cases] page):

```
grieg$ php advisor 3211807 09s2
... suggest subjects for Gerard Rettke to take in 10s1 ...
grieg$ php advisor 3211807 10s1
... suggest subjects for Gerard Rettke to take in 10s2 ...
grieg$ php advisor 1234567 09s1
Invalid student (1234567)
grieg$ php advisor 3211807 20s1
Invalid term (20s1)
```

The `advisor` script gets its data from a function (`advice()`). The result of this function is an array of possible subjects to enrol in in the semester following the specified one. Information about each subject is provided as an array, with the following elements:

- subject code
- subject name (i.e. `Subject.name`)
- UOC which will be awarded for this subject
- which rule it aims to satisfy (using `ruleName()`)

Some examples of possible elements in the array returned by `advice()`:

```
array("COMP1917","Computing 1A",6,"Level 1 Core")
array("COMP2041","Software Construction",6,"Level 2 Core")
array("COMP3311","Database Systems",6,"Level 3/4 Electives")
```

The array returned by the `advice()` function should only include subjects satisfying the following:

- they are are available in the next semester
- the student satisfies the pre-requisites**
- they are not in the exclusion or equivalence group of any course already taken
- they would contribute towards the student completing their degree
- they would not cause the maxuoc of any relevant limit rule to be exceeded

Note that pre-reqs are associated with a career (UG or PG) and the only the pre-reqs relevant to the student (determined via program enrolment) need to be applied. If a course has no pre-reqs, or no pre-reqs for the career of the student, then a student can only take the course if it matches their career. For example, a UG student cannot take a PG course unless it has some UG pre-reqs.

In determining whether a student can take a course, make the following assumptions:

- it is week 9 of the semester, so that all withdrawals, etc. from courses in the current semester have been finalised (i.e. `NF` for withdrawals before week 8, `DF` for withdrawals after week 8)
- students will pass all of the subjects they're currently enrolled in
  (except any with `NF` grade, which can be ignored, and any with `DF`, which count as fails
- the "next" semester from S1 is S2 and the "next" semester from S2 is S1 of the following year (i.e. ignore summer and winter terms); the supplied `nextSemester()` function can be used to determine this
- future semesters (those in the database, but with no courses data) will have the same course offerings as the corresponding semester in the most recent year which has courses data (e.g. 14s1, will use the courses for 13s1; 14s2 etc. will use the courses for 13s2)

Use the transcript results up to and including those for the current semester (i.e. the results generated by `transcript(int,int)`).

In order to give advice, you need to know the program/streams that the student is enrolled in. If the student has a program/stream enrolment for the current semester, use that (assume that they won't change enrolment). If the student is not enrolled in the current semester, but has been enrolled in the past, use the most recent program enrolment. For the special case of a first-year student just starting out, if you specify a semester that is earlier than any of the student's enrolled semesters, then use the student's program enrolment for the "next" semester. If you cannot determine the program/stream enrolment, then simply return an empty advice array.

The array should be generated in "priority order":

- core courses rank highest
- courses from "professional elective" groups rank next
- Free Electives rank next
- General Education subjects rank lowest

within each group, courses should be ordered by their subject code.

Since there will typically be a large number of possible General Education courses that a student might take, and an even larger number of Free Electives, the array contains a single item to summarise each of these groups. These special items should indicate which group of subjects is involved, ~~the total number of subjects available for the student in that group~~**, and the number of UOC from the group that the student needs to take to complete their requirement. For example, if they had already take 9UOC of General Education subjects, then the system would show that they only needed to take 3UOC of Gen Ed, and would not include any 6UOC Gen Ed subjects in the count of available courses.

The following examples show what these special items should look like:

```
array("GenEd...","General Education (many choices)",9,"Gen Ed"),
array("Free....","Free Electives (many choices)",18,"Free Electives"),
```

In the above example, we assume that the student has taken just one 3UOC subject towards their 12UOC Gen Ed requirement and 30UOC of courses towards their 48UOC Free Elective requirement.

** The spec originally asked you to count the number of GenEds or Free Electives. Since there are so many subjects to check, this turns out to take too long, so it is no longer required to count the Gen Eds or Free Electives.

Since `LR` rules with a `min` UOC tend to be associated with large subject groups, you should treat them in a similar way to free electives and gen eds.

```
array("Limit...","Rule Name (many choices)",UOC,"Rule Name")
```

where *UOC* is the number of UOC still required to reach the `min` value for the rule.

Note that it is remotely possible that some of the subject groups or stream groups used in pre-req rules (type=RQ) are empty. If the subject pre-req involves such an empty group, simply assume that the student meets the pre-req.

Note that no `MR` rules in the database are associated with a subject or stream group. They are all of the simple form "must have completed *X* UOC". You can assume that this will also be true in the testing database.

You need to implement the `advice()` function used by the `advisor` script to generate the suggested courses array This function is defined as:

```
function advice($db, $studentID, $currTermID, $nextTermID)
```

where the parameters are:

- `$db` ... an open database connection
- `$studentID` ... the `People.id` value for the student (*not* their UNSWid)
- `$currTermID` ... the `Terms.id` value for the specified term
- `$nextTermID` ... the `Terms.id` value for the following term

You should place definition of this function in the `ass3.php` file. Any SQL or PLpgSQL functions that you write to support it should be placed in the `updates.sql` file.

Hint: start similar to Task D; then generate a list of courses for the next term; check whether they can take each course; allocate it to a not-yet-complete requirement; alternatively, work from the requirements to see what courses are needed and then see if those courses are offered.

Have fun, *jas*