COMP3311 19T3 Database Systems

Exercises 05
Stored Functions in SQL and PLpgSQL

1. Write a simple PLpgSQL function that returns the square of its argument value. It is used as follows:

```
mydb=> select sqr(4);
 sqr
-----
  16
(1 row)

mydb=> select sqr(1000);
   sqr
---------
 1000000
(1 row)
```

Could we use this function to square real numbers? If not, how could we write a function to achieve this?

[hide answer]

```
create or replace function sqr(n integer) returns integer
as $$
declare
        result integer;
begin
        result := n * n;
        return result;
end;
$$ language plpgsql;
```

OR

```
create or replace function sqr(n integer) returns integer
as $$
begin
        return n * n;
end;
$$ language plpgsql;
```

This function won't square real numbers, even something like:

```
mydb=> select sqr(3.0);
ERROR:  Function sqr(numeric) does not exist
```

The types don't match, and PostgreSQL won't automatically convert. However, by declaring the function to use the generic number type `numeric`, it will handle both integers and reals correctly:

```
create or replace function sqr(n numeric) returns numeric
as $$
begin
        return n * n;
end;
$$ language plpgsql;
```

2. Write two PLpgSQL functions to compute factorial, one iterative and one recursive. Both functions take an integer argument and return an integer result, if the argument is non-negative. Otherwise, they return `NULL`. They would both be used as follows:

```
mydb=> select fac(5);
 fac
-----
 120
```

```
(1 row)

mydb=> select fac(10);
   fac
---------
 3628800
(1 row)

mydb=> select fac(0);
 fac
-----
   1
(1 row)

mydb=> select fac(-1);
 fac
-----
(0 rows)
```

[hide answer]

```
-- recursive version
create or replace function fac(n integer) returns integer
as $$
begin
        if (n < 0) then
                return null;
        elsif (n = 0) then
                return 1;
        elsif (n = 1) then
                return 1;
        else
                return n * fac(n - 1);
        end if;
end;
$$ language plpgsql;

-- iterative version
create or replace function fac(n integer) returns integer
as $$
declare
        i integer;
        res integer;
begin
        if (n < 0) then
                return null;
        end if;
        res := 1;
        for i in 1 .. n loop
                res := res * i;
        end loop;
        return res;
end;
$$ language plpgsql;
```

3. Write a PLpgSQL function that "spreads" the letters in some text. It is used as follows:

```
mydb=> select spread('My Text');
     spread
----------------
 M y   T e x t
(1 row)
```

[hide answer]

```
create or replace function spread(text) returns text
as $$
```

```
declare
        result text := '';
        i      integer;
        len    integer;
begin
        i := 1;
        len := length($1);
        while (i <= len) loop
                result := result || substr($1, i, 1) || ' ';
                i := i+1;
        end loop;
        return result;
end;
$$ language plpgsql;
```

OR

```
create or replace function spread(text) returns text
as $$
declare
        result text := '';
        i      integer;
begin
        i := 1;
        for i in 1..length($1) loop
                result := result || substr($1, i, 1) || ' ';
        end loop;
        return result;
end;
$$ language plpgsql;
```

Note that if you omit the initial assignment of empty string to `result`, then the value of `result` stays as NULL throughout the entire function execution, and NULL is returned (i.e., string concatenation is a null-preserving operation).

4. Write a PLpgSQL function to return a table of the first *n* positive integers. You may assume the existence of a tuple type:

```
create type IntValue as ( val integer );
```

The function is used as follows:

```
mydb=> select * from seq(5);
 val
-----
   1
   2
   3
   4
   5
(5 rows)
```

and has the following signature:

```
create or replace function seq(int) returns setof IntValue
```

[hide answer]

```
create or replace
        function seq(int) returns setof IntValue
as $$
declare
        i integer;
        r IntValue%rowtype;
begin
        for i in 1 .. $1
        loop
```

```
                r.val = i;
                return next r;
        end loop;
        return;
end;
$$ language plpgsql;
```

5. Generalise the previous function so that it returns a table of integers, starting from *lo* up to at most *hi*, with an increment of *inc*. The function should also be able to count down from *lo* to *hi* if the value of *inc* is negative. An *inc* value of 0 should produce an empty table. Use the following function header:

```
create or replace function seq(int,int,int) returns setof IntValue
```

and the function would be used as follows:

```
mydb=> select * from seq(2,7,2);
 val
-----
  2
  4
  6
(3 rows)
```

Some other examples, in a more compact representation:

```
seq(1,5,1)   gives   1  2  3  4  5
seq(5,1,-1)  gives   5  4  3  2  1
seq(9,2,-3)  gives   9  6  3
seq(2,9,-1)  gives   empty
seq(1,5,0)   gives   empty
```

[hide answer]

```
create or replace function
        seq(lo int, hi int, inc int) returns setof IntValue
as $$
declare
        i integer;
        r IntVal%rowtype;
begin
        i := lo;
        if (inc > 0) then
                while (i <= hi)
                loop
                        r.val = i;
                        return next r;
                        i := i + inc;
                end loop;
        elsif (inc < 0) then
                while (i >= hi)
                loop
                        r.val = i;
                        return next r;
                        i := i + inc;
                end loop;
        end if;
        return;
end;
$$ language plpgsql;
```

6. Re-implement the `seq(int)` function from above as an **SQL function**, and making use of the generic `seq(int,int,int)` function defined above.

[hide answer]

```
create or replace function
        seq(n int) returns setof IntValue
```

```
as $$
declare
        r IntVal%rowtype;
begin
        for r in select * from seq(1,n,1)
        loop
                return next r;
        end loop;
        return;
end;
$$ language plpgsql;
```

or, it could be done more simply as an SQL function:

```
create or replace function
        seq(int) returns setof IntValue
as $$
        select * from seq(1,$1,1);
$$
language sql;
```

7. Create a new version of the factorial function based on the above sequence returning functions. Implement it as an **SQL function** (not a PLpgSQL function). The obvious solution to this problem requires a `product` aggregate, analogous to the `sum` aggregate. Since PostgreSQL does not have a built-in product aggregate, you will need to build your own. In order to achieve the same behaviour as the above PLpgSQL function for non-positive numbers, you will need to use the same function signature as before, i.e.:

   ```
   create or replace function fac(int) returns int
   ```

   Note that PostgreSQL does *not* have a product aggregate, so you will need to build your own.

   [hide answer]

   ```
   create function fac(int) returns int
   as $$
           select product(val) from seq($1);
   $$
   language sql;

   -- where the product aggregate is defined as follows:
   -- basetype = type of the input values
   -- stype    = type of intermediate states in computing aggregate
   -- sfunc    = function mapping (currentState, nextValue) -> newState
   -- initcond = value for starting state
   -- The "intermediate state" for computing a product is very simple.
   -- It's simply the product accumulated so far. Similarly, the mapping
   -- function simply multiplies the current product by the next value.
   --
   -- For more details on defining aggregates, see ...
   --     PostgreSQL Reference Manual, SQL section

   create aggregate product(int) (
           sfunc     = multiplyNext,
           stype     = int,
           initcond  = 1
   );

   create function multiplyNext(int,int) returns int
   as $$
   begin return $1 * $2; end;
   $$ language plpgsql;

   -- note that the above definitions need to be given in the reverse
   -- order to what they're given here in order for this to work
   ```

Use the Beers/Bars/Drinkers database from lectures in answering the following questions. A summary schema for this database:

```
Beers(name:string, manufacturer:string)
Bars(name:string, address:string, license#:integer)
Drinkers(name:string, address:string, phone:string)
Likes(drinker:string, beer:string)
Sells(bar:string, beer:string, price:real)
Frequents(drinker:string, bar:string)
```

The examples below assume that the user is connected to a database called `beer` containing an instance of the above schema.

8. Write a PLpgSQL function called `hotelsIn()` that takes a single argument giving the name of a suburb, and returns the names of all hotels in that suburb, one per line. It is used as follows:

```
beer=> select hotelsIn('The Rocks');
              hotelsin
 ------------------------------
  Australia Hotel
  Lord Nelson

 (1 row)
```

Note that the output from functions returning `text` looks much better if you turn off output alignment (via `psql`'s **`\a`** command) and get rid of column headings (via `psql`'s **`\t`** command).

Compare the aligned output above to the unaligned output below:

```
beer=> \a
Output format is unaligned.
beer=> \t
Showing only tuples.
beer=> select hotelsIn('The Rocks');
Australia Hotel
Lord Nelson
```

From now on, sample outputs for functions returning `text` will assume that we have used **`\a`** and **`\t`**.

[hide answer]

Note that this returns a string and not a list of records.

```
create or replace function
        hotelsIn(_addr text) returns text
as $$
declare
        r   record;
        out text := '';
begin
        for r in select * from bars where addr = _addr
        loop
                out := out || r.name || e'\n';
        end loop;
        return out;
end;
$$ language plpgsql;
```

9. Write a new PLpgSQL function called `hotelsIn()` that takes a single argument giving the name of a suburb and returns the names of all hotels in that suburb. The hotel names should all appear on a single line, as in the following examples:

```
beer=> select hotelsIn('The Rocks');
Hotels in The Rocks:  Australia Hotel  Lord Nelson

beer=> select hotelsIn('Randwick');
Hotels in Randwick:  Royal Hotel

beer=> select hotelsIn('Rendwick');
Hotels in Rendwick:
```

[hide answer]

```
create or replace function
        hotelsIn (_addr text) returns text
as $$
declare
        pubnames text;
        p record;
begin
        pubnames:= 'Hotels in ' || address || ':';
        for p in select * from Bars where addr = _addr
        loop
                pubnames := pubnames||' '||p.name;
        end loop;
        pubnames := pubnames||e'\n';
        return pubnames;
end;
$$ language plpgsql;
```

10. Modify the PLpgSQL function in the previous question so that it prints a more sensible message if there are no hotels in the named suburb. It should also display the hotel names on separate lines and numbers them. It is used as follows:

```
beer=> select hotelsIn('Rendwick');
There are no hotels in Rendwick

beer=> select hotelsIn('The Rocks');
Hotels in The Rocks:
  1. Australia Hotel
  2. Lord Nelson
```

Use `to_char(i,99)` to format the numbers.

[hide answer]

This uses "select ... into" to extract a count.

```
create or replace function
         hotelsIn(_addr text) returns text
as $$
declare
        pubnames text;
        p record;
        i integer := 0;
        howmany integer := 0;
begin
        select count(*) into howmany from Bars where addr = _addr;
        if (howmany = 0) then
                return 'There are no hotels in '|| address ||e'\n';
        end if;
        pubnames := 'Hotels in ' || address || e':\n';
        for p in select * from Bars where addr = _addr
        loop
                i := i+1;
                pubnames := pubnames|| to_char(i,99) ||'. '||p.name||e'\n';
        end loop;
        return pubnames;
end;
$$ language plpgsql;
```

11. Write a PLpgSQL procedure `happyHourPrice` that accepts the name of a hotel, the name of a beer and the number of dollars to deduct from the price, and returns a new price. The procedure should check for the following errors:

    - non-existent hotel (invalid hotel name)
    - non-existent beer (invalid beer name)
    - beer not available at the specified hotel
    - invalid price reduction (e.g. making reduced price negative)

Use `to_char(price,'$9.99')` to format the prices.

```
beer=> select happyHourPrice('Oz Hotel','New',0.50);
There is no hotel called 'Oz Hotel'.

beer=> select happyHourPrice('Australia Hotel','Newer',0.50);
There is no beer called 'Newer'.

beer=> select happyHourPrice('Australia Hotel','New',0.50);
The Australia Hotel does not serve New

beer=> select happyHourPrice('Australia Hotel','Burragorang Bock',4.50);
Price reduction is too large; Burragorang Bock only costs $ 3.50

beer=> select happyHourPrice('Australia Hotel','Burragorang Bock',1.50);
Happy hour price for Burragorang Bock at Australia Hotel is $ 2.00
```

[hide answer]

This could be done using either all `select count(*)` followed by a check for zero, or by using the FOUND variable (which is set after each query). This solution combines both approaches to show the range of possiblities.

In general, you could use `count(*)` whenever you knew that you were not interested in collecting any other information from the table; you'd try to collect the information and use FOUND in all other circumstances.

```
-- using positional notation for parameters
create or replace function
        happyHourPrice (_hotel text, _beer text, _discount real) returns text
as $$
declare
        counter integer;
        std_price real;
        new_price real;
begin
        select into counter count(*) from Bars where name = _hotel;
        if (counter = 0) then
                return 'There is no hotel called '|| _hotel ||e'.\n';
        end if;
        select into counter count(*) from Beers where name = _beer;
        if (counter = 0) then
                return 'There is no beer called '|| _beer ||e'.\n';
        end if;
        select price into std_price
        from   Sells s
        where  s.beer = _beer and s.bar = _hotel;
        if (not found) then
                return 'The '|| _hotel || ' does not serve '||_beer;
        end if;
        new_price := std_price - _discount;
        if (new_price < 0) then
                return 'Price reduction is too large; '
                        || _beer || ' only costs '
                        || to_char(std_price, '$9.99');
        else
                return 'Happy hour price for '
                        || _beer || ' at '|| _hotel ||' is '
                        || to_char(new_price, '$9.99');
        end if;
end;
$$ language plpgsql;
```

12. The `hotelsIn` function above returns a formatted string giving details of the bars in a suburb. If we wanted to return a table of records for the bars in a suburb, we could use a view as follows:

```
beer=> create or replace view HotelsInTheRocks as
    -> select * from Bars where addr = 'The Rocks';
CREATE VIEW
```

```
beer=> select * from HotelsInTheRocks;
      name       |   addr    | license
-----------------+-----------+---------
 Australia Hotel | The Rocks |  123456
 Lord Nelson     | The Rocks |  123888
(2 rows)
```

Unfortunately, we need to specify a suburb in the view definition. It would be more useful if we could define a "parameterised view" which we could use to generate a table for any suburb, e.g.

```
beer=> select * from HotelsIn('The Rocks');
      name       |   addr    | license
-----------------+-----------+---------
 Australia Hotel | The Rocks |  123456
 Lord Nelson     | The Rocks |  123888
(2 rows)
beer=> select * from hotelsIn('Coogee');
       name       |  addr  | license
------------------+--------+---------
 Coogee Bay Hotel | Coogee |  966500
(1 row)
```

Such a parameterised view can be implemented via an SQL function, defined as:

```
create or replace function hotelsIn(text) returns setof Bars
as $$ ... $$ language sql;
```

Complete the definition of the SQL function.

[hide answer]

```
create or replace function
        hotelsIn(text) returns setof Bars
as $$
        select * from Bars where addr = $1;
$$ language sql;
```

13. The function for the previous question can also be implemented in PLpgSQL. Give the PLpgSQL definition. It would be used in the same way as the above.

[hide answer]

```
create or replace function
        hotelsIn(_addr text) returns setof Bars
as $$
declare
        r record;  -- could also be declared r Bars%rowtype;
begin
        for r in select * from Bars where addr = _addr
        loop
           return next r;
        end loop;
        return;
end;
$$ language plpgsql;
```

Use the Bank Database in answering the following questions. A summary schema for this database:

```
Branches(location:text, address:text, assets:real)
Accounts(holder:text, branch:text, balance:real)
Customers(name:text, address:text)
Employees(id:integer, name:text, salary:real)
```

The examples below assume that the user is connected to a database called `bank` containing an instance of the above schema.

14. For each of the following, write both an SQL and a PLpgSQL function to return the result:

a. salary of a specified employee

[hide answer]

```
-- Salary of a specified employee
--   Allows employee to be determined by name or id
--    using overloading on the function name
--   Assume name or id identifies only one employee

create or replace function empSal(text) returns real
as $$
        select salary from employees where name = $1
$$ language sql;

create or replace function empSal(integer) returns real
as $$
        select salary from employees where id = $1
$$ language sql;

create or replace function
        empSal1(_name text) returns real
as $$
declare
        _sal real;
begin
        select salary into _sal
        from employees where name = _name;
        return _sal;
end;
$$ language plpgsql;

create or replace function
        empSal1(_id integer) returns real
as $$
declare
        _sal real;
begin
        select salary into _sal
        from employees where id = _id;
        return _sal;
end;
$$ language plpgsql;
```

b. all details of a particular branch

[hide answer]

```
-- All details of a particular branch
--   Example of PLpgSQL function returning a record

create or replace function branchDetails(text) returns Branches
as $$
        select * from Branches where location = $1;
$$ language sql;


create or replace function branchDetails1(_bname text) returns Branches
as $$
declare
        _tup Branches;
begin
        select * into _tup
        from Branches where location = _bname;
        return _tup;
end;
$$ language plpgsql;
```

c. names of all employees earning more than $*sal*

[hide answer]

```
-- Names of all employees earning more than $sal
--   Example of PLpgSQL function returning a set of atomic values

create or replace function empsWithSal(real) returns setof text
as $$
        select name from employees where salary > $1;
$$ language sql;

create type EmpName as ( name text );

create or replace function empsWithSal1(_minSal real) returns setof EmpName
as $$
declare
        _en EmpName;
begin
        for _en in select name
                   from employees where salary > _minSal
        loop
                return next _en;
        end loop;
        return;
end;
$$ language plpgsql;
```

d. all details of highly-paid employees

[hide answer]

```
-- All details of highly-paid employees
--   Example of PLpgSQL function returning a set of atomic values

create or replace function richEmps(real) returns setof Employees
as $$
        select * from employees where salary > $1;
$$ language sql;

create or replace function emps1(_minSal real) returns setof Employees
as $$
declare
        _e Employee;
begin
        for _e in select *
                  from employees where salary > _minSal
        loop
                return next _e;
        end loop;
        return;
end;
$$ language plpgsql;
```

15. Write a PLpgSQL function to produce a report giving details of branches:

    - name and address of branch
    - list of customers who hold accounts at that branch
    - total amount in accounts held at that branch

    Use the following format for each branch:

```
Branch: Clovelly, Clovelly Rd.
Customers:  Chuck Ian James
Total deposits: $   8860.00
```

[hide answer]

```
create or replace function branchList() returns text
as $$
declare
```

```
        a   record;
        b   record;
        tot integer;
        qry text;
        out text := e'\n';
begin
        for b in select * from Branches
        loop
                out := out || 'Branch: ' || b.location || ', ';
                out := out || b.address || e'\n' || 'Customers: ';
                tot := 0;
                for a in select * from Accounts where branch=b.location
                loop
                        out := out || ' ' || a.holder;
                        tot := tot + a.balance;
                end loop;
                select sum(balance) into tot
                from Accounts where branch=b.location;
                out := out || E'\nTotal deposits: ';
                out := out || to_char(tot,'$999999.99');
                out := out || E'\n---\n';
        end loop;
        return out;
end;
$$ language plpgsql;
```

It's also possible to implement this more efficiently using just one SQL query (rather than nested-loop queries). The more efficient solution invloves ordering the Accounts tuples by branch, and keeping track of when the current branch changes to a new one.

Use the UNSWSIS Database in answering the following questions. Note that the UNSWSIS schema is very similar to *but not the same as* the MyMyUNSW schema (in fact, UNSWSIS is an older version of MyMyUNSW). The schema is too large to give a complete summary here, but we provide some details for some tables:

```
Person(id:integer, ..., name:text, ...)
Student(id:integer, sid:integer, stype:('local','intl'))
Staff(id:integer, sid:integer, office:integer, ...)
Term(id:integer, year:integer, session:('S1','S2','X1','X2'), ...)
Subject(id:integer, code:text, ..., name:text, ... uoc:integer, ...)
Course(id:integer, subject:integer, term:integer, lic:integer, ...)
```

The examples below assume that the user is connected to a database called `unswsis` containing an instance of the above schema.

16. Write a PLpgSQL function to produce the complete name of an OrgUnit:

```
function unitName(_ouid integer) returns text
```

The function returns the complete name using the rules:

- the university is denoted by UNSW
- a faculty is denoted using its base name (not all faculty names start with Faculty)
- a school is denoted School of XYZ
- a department is denoted Department of XYZ
- a centre is denoted Centre for XYZ
- an institute is denoted Institute of XYZ
- other kinds of OrgUnits are treated as having no name (i.e. return null)

[hide answer]

```
create or replace function unitName(_ouid integer) returns text
as $$
declare
        _outype text;
        _ouname text;
begin
        -- check whether the orgunit ID is valid
        select * from OrgUnit where id = _ouid;
```

```
        if (not found) then
                raise exception 'No such unit: %',_ouid;
        end if;

        select t.name,u.longname into _outype,_ouname
        from   OrgUnitType t, OrgUnit u
        where  u.id = _ouid and u.utype = t.id;

        -- debugging output
        -- raise notice 'Type:%, Name:%',_outype,_ouname;

        if (_outype = 'UNSW') then
                return 'UNSW';
        elsif (_outype = 'Faculty') then
                return _ouname;
        elsif (_outype = 'School') then
                return 'School of '||_ouname;
        elsif (_outype = 'Department') then
                return 'Department of '||_ouname;
        elsif (_outype = 'Centre') then
                return 'Centre for '||_ouname;
        elsif (_outype = 'Institute') then
                return 'Institute of '||_ouname;
        else
                return null;
        end if;
end;
$$ language plpgsql;
```

An alternative, using an SQL `CASE` expression:

```
create or replace function unitName(_ouid integer) returns text
as $$
declare
        _ouname text;
begin
        -- check whether the orgunit ID is valid
        select * from OrgUnit where id = _ouid;
        if (not found) then
                raise exception 'No such unit: %',_ouid;
        end if;

        select case
               when t.name = 'UNSW' then 'UNSW'
               when t.name = 'Faculty' then t.longname
               when t.name = 'School' then 'School of '||t.longname
               when t.name = 'Department' then 'Department of '||t.longname
               when t.name = 'Centre' then 'Centre for '||t.longname
               when t.name = 'Institute' then 'Institute of '||t.longname
               else null
               end into _ouname
        from   OrgUnitType t, OrgUnit u
        where  u.id = _ouid and u.utype = t.id;
        return _ouname;
end;
$$ language plpgsql;
```

If you didn't care about error-checking on the OrgUnit ID, then this could be done as an SQL function.

17. Write a PLpgSQL function to produce a transcript as a set of `TranscriptEntry` tuples, where the tuple type is defined as:

```
create type TranscriptEntry as (
   course  text,      -- e.g. 'COMP1911 06s1 Computing 1'
   mark    integer,   -- e.g. 50
   grade   char(2),   -- e.g. 'PS'
```

```
   uoc     integer    -- e.g. 6
);
```

As well as producing the standard `TranscriptEntry` tuples, this function should also produce a "special" `TranscriptEntry` tuple at the end of each run of courses from a given session, where:

- the `course` field holds the string `'WAM for YYsN'`
- the `mark` field holds the WAM for that session
- all other attributes are NULL

The WAM (weighted average mark) is an important summary of a student's achievement at UNSW. Let us assume that WAMs are computed according to the following:

- each course for which a mark is available is counted, all other courses are ignored (e.g. courses graded SY/FL have no mark and are thus not included)
- General Education is counted as a normal course for computing the WAM
- substitutions or advanced standing subjects are counted as normal courses for computing the WAM, provided that they are based on UNSW subjects
- courses with marks available are denoted c1, c2, c3, ...
- the units of credit for course c1 are denoted uoc(c1)
- weighted_sum = mark(c1)*uoc(c1) + mark(c2)*uoc(c2) + ...
- WAM = weighted_sum / (uoc(c1) + uoc(c2) + ...)

The WAM should be computed while the course result tuples are being read; it should not be computed by invoking a new query.

[hide answer]

```
-- result tuples for internal ts() function

create type TranscriptRecord as (
        code    char(8),
        title   text,
        term    integer,
        year    integer,
        sess    char(2),
        tstart  date,
        mark    integer,
        grade   char(2),
        uoc     integer
);

-- ts() returns a list of Transcript tuples for one student

create or replace function
        ts(integer) returns setof TranscriptRecord
as $$
select s.code,s.name as title,
        t.id as term,t.year,t.sess,t.startdate as tstart,
        e.mark,e.grade,s.uoc
from   Subject s, Course c, Term t,
        CourseEnrolment e, Student stu
where  stu.sid = $1 and stu.id = e.student
        and e.course = c.id and c.term = t.id
        and c.subject = s.id
$$
language sql;


--  result tuples for transcript() function

create type TranscriptEntry as (
        course  text,      -- e.g. 'COMP1911 06s1 Computing 1'
        mark    integer,   -- e.g. 50
        grade   char(2),   -- e.g. 'PS'
        uoc     integer    -- e.g. 6

-- builds contents of special TranscriptEntry for term WAM
```

```
create or replace function
        termRec(_tr TranscriptRecord, _weighted numeric, _sumUoC integer)
        returns TranscriptEntry
as $$
declare
        _te  TranscriptEntry;
begin
        _te := (null,null,null,null);
        _te.course := 'WAM for ' ||termName(_tr.year,_tr.sess);
        if (_sumUoC = 0) then
                _te.course = 'No '||_te.course;
        else
                _te.mark = (_weighted/_sumUoC)::integer;
        end if;
        _te.course := '=== '||_te.course;
        return _te;
end;
$$ language plpgsql;

-- main transcript() function that drives everything else

create or replace function
        transcript(_sid integer) returns setof TranscriptEntry
as $$
declare
        _stu          Student;
        _termWeighted numeric := 0;
        _termSumUoC   integer := 0;
        _tr           TranscriptRecord;
        _prev         TranscriptRecord;
        _te           TranscriptEntry;
begin
        select * into _stu from Student where sid = _sid;
        if (not found) then
                raise exception 'No such student: %',_sid;
        end if;

        -- foreach transcript record, in the correct order

        for _tr in select * from ts(_sid) order by tstart,code
        loop
                if (_prev.term <> _tr.term) then
                        -- transition from one term to next
                        -- summarise any existing term info
                        if (_prev.term is not null) then
                                _te := termRec(_prev,_termWeighted,_termSumUoC);
                                return next _te;
                        end if;
                        -- set up to compute wam for next term
                        _termWeighted := 0;
                        _termSumUoc := 0;
                end if;
                _te.course := _tr.code||' '||termName(_tr.year,_tr.sess)
                                ||' '||_tr.title;
                _te.mark   := _tr.mark;
                _te.grade  := _tr.grade;
                _te.uoc    := _tr.uoc;
                return next _te;
                if (_tr.mark is not null) then
                        _termWeighted := _termWeighted + (_tr.mark * _tr.uoc);
                        _termSumUoC := _termSumUoC + _tr.uoc;
                end if;
                _prev := _tr;
        end loop;
        if (_prev.term is not null) then
                _te := termRec(_prev,_termWeighted,_termSumUoC);
                return next _te;
        end if;
```

```
        return;
end;
$$ language plpgsql;
```

18. Write a PLpgSQL function which takes the numeric identifier of a given OrgUnit and returns the numeric identifier of the parent faculty for the specified OrgUnit:

```
function facultyOf(_ouid integer) returns integer
```

Note that a faculty is treated as its own parent. Note also that some OrgUnits don't belong to any faculty; such OrgUnits should return a null result from the function.

[hide answer]

```
create or replace function facultyOf(_ouid integer) returns integer
as $$
declare
        _count integer;
        _tname text;
        _parent integer;
begin
        select count(*) into _count
        from orgUnit where id = _ouid;
        if (_count = 0) then
                raise exception 'No such unit: %',_ouid;
        end if;

        select t.name into _tname
        from OrgUnit u, OrgUnitType t
        where u.id = _ouid and u.utype = t.id;

        if (_tname is null) then
                return null;
        elsif (_tname = 'University') then
                return null;
        elsif (_tname = 'Faculty') then
                return _ouid;
        else
                select owner into _parent
                from UnitGroups where member = _ouid;
                return facultyOf(_parent);
        end if;
end;
$$ language plpgsql;
```

An alternative way of checking the existence of the specified organisational unit would be:

```
        select * from OrgUnit where id = _ouid;
        if (not found) then
                raise exception 'No such unit: %',_ouid;
        end if;
```