Laboratory 5 Variables, Functions and Procedures

Development and Editing: Z. Garofalaki, A. Tsolakidis, P. Andritsos

Target

Familiarity with the definition and use of variables, the creation of functions and procedures

Tools

A. Variables

1. Variable creation is done with the following general command form. Each variable and its content are preserved for the user's specific session.

```
SET @var_name = expr [, @var_name = expr] ...
```

2. The **creation of variables** with: name alpha and value 10, name beta and value 20, is done with the command:

```
set @alpha=10, @beta=20;
```

3. Displaying the contents of a variable is done with the command:

```
select @alpha;                           //display value of variable @alpha

select @beta;                            //display value of variable @beta

select @chi:=@alpha+@beta; //display value of variable @chi which is the sum of
                                         //alpha and beta
```

Activities

Carry out the following activities. The command or commands required for the implementation of each step, as well as the result of its execution should be included in a deliverable file in text or screenshot format. The file or files with your answers should be compressed into a **xx_ÿÿÿÿÿ_EPONYMO.zip,** where: (a) xx is the number of the section you belong to (e.g. for group [02] MONDAY 12:00-13 :00, **xx = 02)** and (b) YYYYYY your Registration Number. This final file will be submitted to the e-class -> DATABASES II -> Assignments.

1. **Connect** to your system's MySQL using any of the above methods you wish.

2. Check if there is a DB named **my_accounts.**

3. Create the my_accounts DB, select it to use, and create a table named Accounts with structure and contents as shown in the following commands. Show the result by displaying (a) the table list of the DB, (b) the contents, and (c) the structure of the Accounts table.

```
drop database if exists my_accounts;
create database my_accounts;
use my_accounts;
create table Accounts ( acctID integer not null primary key,
Balance integer not null); insert
into Accounts (acctID, Balance) values (101, 1000); insert into Accounts
(acctID, Balance) values (202, 2000); insert into Accounts (acctID, Balance)
values (303, 2500); insert into Accounts (acctID, Balance) values (404, 3000);
```

4. Display the contents of the Accounts table, adding ascending numbering to records of:

```
set @rownum=0;
select (@rownum:=@rownum+1) as No, acctID, Balance from Accounts order by acctID;
```

5. Should the ascending numbering that appeared in the column titled No in step 4 also exist in the Accounts table? Justify your answer.

6. Add the CUSTOMERS table shown in Figure 1. Set data types of the CUSTNO and CUST_NAME columns to integer and varchar(30) respectively. Show the result by displaying (a) the table list of the NW, (b) the contents, and (c) the structure of the CUSTOMERS table.

7. In the Accounts table add a column named Custno, data type integer and set it as FK of the Accounts table to link its records with the records of the CUSTOMERS table. Update the contents of the Custno column so that the account with AcctID=202 corresponds to customer code 20 and all other accounts correspond to customer code 10. Show the result by displaying (a) the contents and (b) the structure of the Accounts table .

8. Execute and interpret the following SQL statements:

```
select CUSTNO, count(*), sum(Balance) from Accounts

where CUSTNO not in (20) group by
CUSTNO?

//Variation using variable set @CUST_NO=20;


select CUSTNO, count(*), sum(Balance) from Accounts where
CUSTNO not in
(@CUST_NO) group by CUSTNO;
```

9. Execute and interpret the following SQL statements:

```
select count(*), sum(Balance) from Accounts;

//Variation using variables set @COUNT_acctID=0,
@SUM_acctID=0, @AVG_acctID=0;

select count(*), sum(Balance), avg(Balance) into @COUNT_acctID,
@SUM_acctID, @AVG_acctID from Accounts;


select @COUNT_acctID, @SUM_acctID, @AVG_acctID, @MY_AVG := @SUM_acctID/@COUNT_acctID;
```

10. Define and use the factorial function that calculates **n!=1*2*…*n:**

```
drop function if exists factorial? delimiter ! create function
factorial(N int)
returns int deterministic


begin
declare F int default 1; while N > 0 do
set F = N *
                        F?
  set N = N - 1; endwhile?
return F; end!
delimiter ;
```

```
// Tests the trigger function for N=4 and N=15 select factorial(4);


select factorial(15);
```

11. Define and use the procedure my_procedure_Local_Variables for calculations using local variables:

```
drop procedure my_procedure_Local_Variables; delimiter $$ create
procedure
my_procedure_Local_Variables() begin set @X = 25; set @Y = 10; select
@X,
@Y, @X*@Y; end
$$ delimiter ; // Call
the procedure call


my_procedure_Local_Variables();
```

Machine Translated by Google

12. Follow the steps below to create a stored procedure and use it of commit/rollback. Explain what the procedure myProc does.

```
// Tests using the MOD function
SET @p_no=3;
SELECT MOD(@p_no,2);
SET @p_no=8;
SELECT MOD(@p_no,2);

// Create base and table
DROP TABLE IF EXISTS myTrace?
CREATE TABLE myTrace ( t_no INT, t_user
CHAR(20), t_date
DATE, t_time
TIME, t_proc
VARCHAR(16), t_what VARCHAR(30));

// Create stored procedure myProc
DROP PROCEDURE IF EXISTS myProc;
DELIMITER !
CREATE PROCEDURE myProc (IN p_no INT, IN p_in VARCHAR(30), OUT
        p_out VARCHAR(30))
LANGUAGE SQL
BEGIN
SET p_out=p_in; INSERT
INTO myTrace (t_no, t_user, t_date, t_time, t_proc, t_what)
  VALUES (p_no, current_user, current_date, current_time, 'myProc', p_in);
IF (MOD(p_no,2)=0) THEN COMMIT;
   ELSE
ROLLBACK? END IF?
END!

DELIMITER ;

// Call the procedure
SET AUTOCOMMIT=0;
CALL myProc(1, 'hello1', @out);
CALL myProc(2, 'hello2', @out);
CALL myProc(3, 'hello3', @out);
CALL myProc(4, 'hello4', @out);
CALL myProc(5, 'hello5', @out);
CALL myProc(6, 'hello6', @out);
CALL myProc(7, 'hello7', @out);
```

```
// Study result
SELECT * FROM myTrace;
```

13. In the Accounts table (Figure 1) the transfer of money from one account to another could be implemented with two UPDATE statements. Here is an example of solving using a transaction. This transaction is characterized as unreliable, as no check is made regarding: (a) the existence of the account to which the money is transferred and (b) the adequacy of the account from which the money is transferred.

```
// Create an Accounts table with 2 entries
DROP TABLE IF EXISTS Accounts?
CREATE TABLE Accounts ( acctID INTEGER NOT NULL PRIMARY KEY,
          balance INTEGER NOT NULL,
CONSTRAINT unloanable_account CHECK (balance >= 0));
INSERT INTO Accounts (acctID, balance) VALUES (101, 1000);
INSERT INTO Accounts (acctID, balance) VALUES (202, 2000);
COMMIT?
SELECT * FROM accounts;

// Transaction
BEGIN TRANSACTION?
UPDATE Accounts SET balance = balance - 100
WHERE acctId = 101;
UPDATE Accounts SET balance = balance + 100
WHERE acctId = 202;
COMMIT?

// Transaction result
SELECT * FROM accounts;
```

14. Here is a solution to the problem in step 15 using the BankTransfer procedure:

```
// Create procedure BankTrasfer
DELIMITER //
DROP PROCEDURE BankTransfer //
CREATE PROCEDURE BankTransfer (IN fromAcct INT,
                                                        IN toAcct INT,
                                                        IN amount INT,
                                                        OUT msg )        VARCHAR(100)

P1: BEGIN
       DECLARE rows INT ;
       DECLARE newbalance INT;
       SELECT COUNT(*) INTO rows FROM Accounts WHERE acctID = fromAcct;
       UPDATE Accounts SET balance = balance - amount WHERE acctID = fromAcct;
       SELECT balance INTO newbalance FROM Accounts WHERE acctID = fromAcct;
       IF rows = 0 THEN
              ROLLBACK?
              SET msg = CONCAT('rolled back because of missing account ', fromAcct);
       ELSEIF newbalance < 0 THEN
                     ROLLBACK?
            SET msg = CONCAT('rolled back because of negative balance of account ', fromAcct);
              ELSE
                     SELECT COUNT(*) INTO rows FROM Accounts WHERE acctID = toAcct;
                     UPDATE Accounts SET balance = balance + amount WHERE acctID = toAcct;
                     IF rows = 0 THEN
                            ROLLBACK?
                            SET msg = CONCAT('rolled back because of missing account ', toAcct);
                     ELSE
                            COMMIT?
                            SET msg = 'committed';
                     END IF?
              END IF?
END P1 //
DELIMITER ;
```

```
// Test transfer 100 from acctID=101 to acctID=202
SET AUTOCOMMIT=0;
SET @out = ' ';
CALL BankTransfer (101, 202, 100, @out);
SELECT @OUT;
Select * from accounts;
COMMIT?

// Test transfer 100 from acctID=101 to acctID=201 (non-existent)
SET autocommit=0;
SET @out = ' ';
CALL BankTransfer (100, 201, 100, @out);
SELECT @OUT;
Select * from accounts;
COMMIT?

// Test transfer 100 from acctID=100 (non-existent) to acctID=201
SET autocommit=0;
SET @out = ' ';
CALL BankTransfer (100, 201, 100, @out);
SELECT @OUT;

// Test transfer 1500 from acctID=101 (insufficient) to acctID=201
SET AUTOCOMMIT=0;
SET @out = ' ';
CALL BankTransfer (101, 201, 1500, @out);
SELECT @OUT;
Select * from accounts;
COMMIT?
```

15. Here is a second solution to the problem in step 15:

```
// Create Accounts table
DROP TABLE IF EXISTS Accounts?
CREATE TABLE Accounts ( acctID INTEGER NOT NULL PRIMARY KEY,
        balance INTEGER NOT NULL);
INSERT INTO Accounts (acctID, balance) VALUES (101, 1000);
INSERT INTO Accounts (acctID, balance) VALUES (202, 2000);
COMMIT?
SELECT * FROM accounts;

// Create trigger Accounts_upd_trg to control updates
delimiter !
CREATE TRIGGER Accounts_upd_trg BEFORE
UPDATE ON Accounts
FOR EACH ROW
BEGIN
IF NEW.balance < 0 THEN
SIGNAL SQLSTATE '23513'
SET MESSAGE_TEXT = 'Negative balance not allowed'; END IF? END? !
delimiter ;



// Create trigger Accounts_ins_trg to control inserts
delimiter !
CREATE TRIGGER Accounts_ins_trg BEFORE INSERT
ON Accounts
FOR EACH ROW
BEGIN
IF NEW.balance < 0 THEN
SIGNAL SQLSTATE '23513'
SET MESSAGE_TEXT = 'Negative balance not allowed'; END IF? END? !
delimiter ;
```

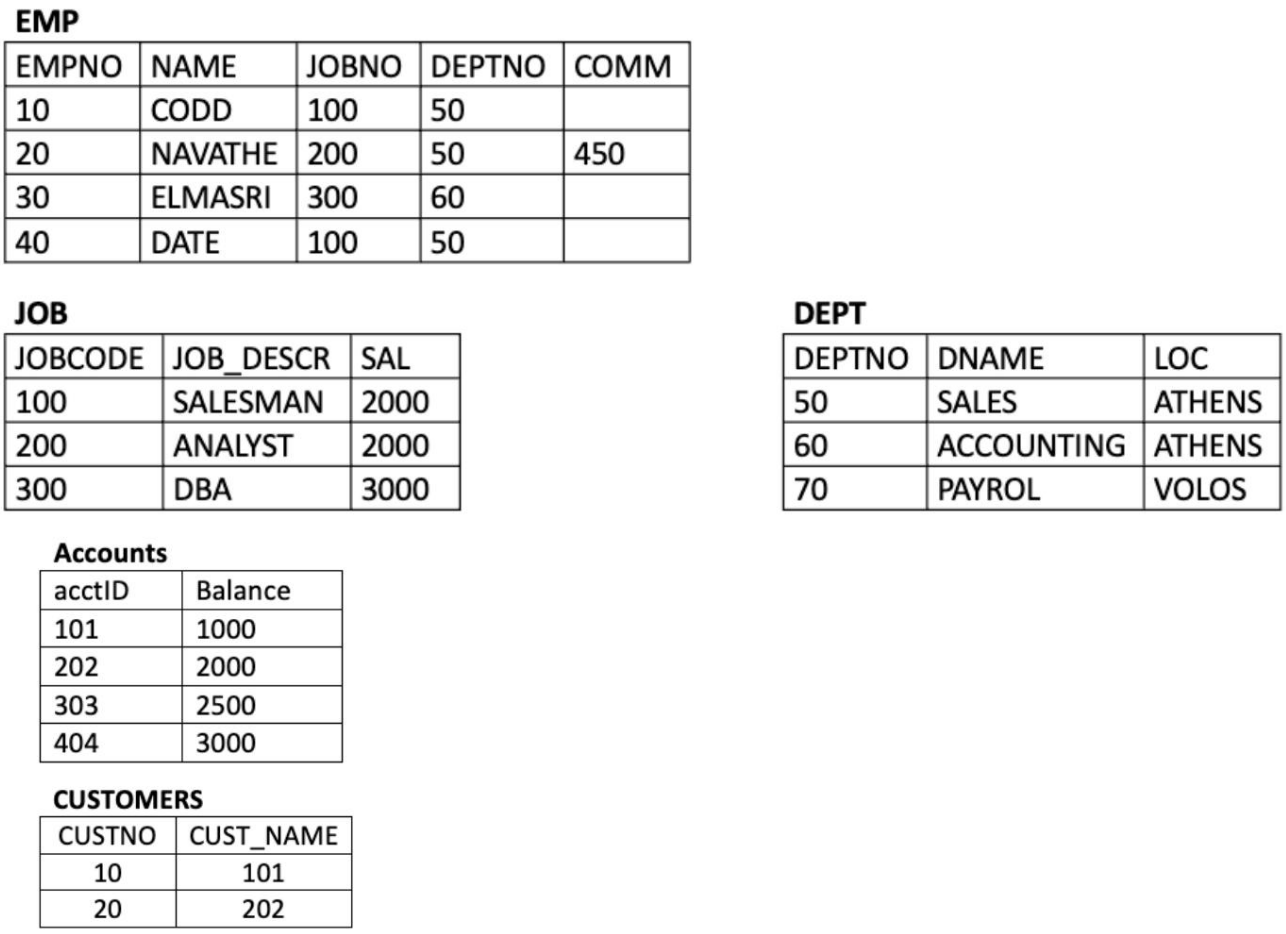
```
// Create procedure BankTransfer
DELIMITER !
CREATE PROCEDURE BankTransfer (IN fromAcct INT,
IN toAcct INT,
IN amount INT,
OUT msg VARCHAR(100))
LANGUAGE SQL MODIFIES SQL DATA
P1: BEGIN
DECLARE acct INT;
DECLARE balance_v INT;
DECLARE EXIT HANDLER FOR NOT FOUND
        BEGIN ROLLBACK? SET
        msg = CONCAT('missing account ', CAST(acct AS CHAR)); END? DECLARE EXIT HANDLER FOR

SQLEXCEPTION BEGIN ROLLBACK; SET msg =
        CONCAT('negative
        balance (?) in ', fromAcct); END? SET acct = fromAcct; SELECT acctID INTO acct FROM Accounts

WHERE acctID = fromAcct ;
UPDATE Accounts SET balance = balance - amount WHERE acctID = fromAcct; SET acct = toAcct; SELECT
acctID INTO acct FROM Accounts WHERE acctID = toAcct ; UPDATE Accounts
SET balance = balance + amount WHERE
acctID = toAcct; SELECT
balance INTO balance_v FROM accounts



    WHERE acctID = fromAcct; IF
      balance_v < 0 THEN
                ROLLBACK?
                SET msg = CONCAT(' negative balance in ', fromAcct);
      ELSE
         COMMIT?
         SET msg = 'committed';
END IF?
END P1 !
DELIMITER ;

CALL BankTransfer (101, 201, 100, @msg);
Select @msg?
CALL BankTransfer (100, 202, 100, @msg);
Select @msg?
CALL BankTransfer (101, 202, 100, @msg);
Select @msg?
CALL BankTransfer (101, 202, 2000, @msg);
Select @msg?
```

NW personnel

The tables contained in the personnel database should have the following structure and contents:

Columns Data type	
DEPT.DEPTNO, EMP.DEPTNO numeric(2)	
DNAME, JOB_DESCR	varchar(24)
LOC char(23)	
JOBCODE, JOBNO	numeric(3)
SAL, COMM	numeric(10,2)
EMPNO	numeric(4)
PROJECT.P_ID	int
PROJECT.P_NAME varchar(255)	

Table 1. Data types of EMP, JOB, DEPT tables



EMP

EMPNO	NAME	JOBNO	DEPTNO	COMM
10	CODD	100	50	
20	NAVATHE	200	50	450
30	ELMASRI	300	60	
40	DATE	100	50	

JOB

JOBCODE	JOB_DESCR	SAL
100	SALESMAN	2000
200	ANALYST	2000
300	DBA	3000

DEPT

DEPTNO	DNAME	LOC
50	SALES	ATHENS
60	ACCOUNTING	ATHENS
70	PAYROL	VOLOS

Accounts

acctID	Balance
101	1000
202	2000
303	2500
404	3000

CUSTOMERS

CUSTNO	CUST_NAME
10	101
20	202

Figure 1. Table data

Annex

The Appendix includes specialized examples of the use of triggers, functions and procedures. Follow them to deepen the corresponding concepts. The procedures you will follow and their results **are NOT required to be submitted** for the laboratory exercise.

1. Create a trigger that connects to an account table and adds the values entered (INSERT statements) as amounts to accounts:

```
// Create db my_accounts with accounts table
DROP DATABASE if exists my_accounts;
CREATE DATABASE my_accounts;
USE my_accounts;
DROP TABLE IF EXISTS accounts?
CREATE TABLE accounts (acct_num INT, amount DECIMAL(12,2));
CREATE TABLE total_bal(total_sum INT);
Insert into total_bal VALUES(0);

// Create trigger calc_sum to check inserts
DROP TRIGGER IF EXISTS calc_sum;
DELIMITER #
CREATE TRIGGER calc_sum
BEFORE INSERT ON accounts
FOR EACH ROW
BEGIN
     DECLARE new_sum INT; SELECT
     total_sum INTO new_sum
     FROM total_bal; SET
     new_sum=new_sum+
   NEW.amount; Update total_bal SET total_sum= new_sum;
   END? #



DELIMITER ;

// Tests
INSERT INTO accounts VALUES(100,1000.50),(101, 2000.50),(102, 1500.00); SELECT * FROM
accounts; SELECT * FROM
total_bal; INSERT INTO accounts
VALUES(103,1000.50),(104, 2000.50),(105, 1500.00); SELECT * FROM accounts; SELECT * FROM
total_bal;
```

2. Alternative solution:

```
// Create db my_accounts with accounts table
DROP DATABASE if exists my_accounts;
CREATE DATABASE my_accounts;
USE my_accounts;
DROP TABLE IF EXISTS accounts?
CREATE TABLE accounts (acct_num INT, amount DECIMAL(12,2));
CREATE TABLE total_bal(total_sum INT);
Insert into total_bal VALUES(0);

// Create trigger calc_sum to check inserts
DROP TRIGGER IF EXISTS calc_sum;
DELIMITER #
CREATE TRIGGER calc_sum
BEFORE INSERT ON accounts
FOR EACH ROW
BEGIN
   Update total_bal
```

```
    SET total_sum = total_sum + NEW.amount; END? # DELIMITER ;




// Tests
INSERT INTO accounts VALUES(100,1000.50),(101, 2000.50),(102, 1500.00); SELECT * FROM
accounts; SELECT * FROM
total_bal; INSERT INTO accounts
VALUES(103,1000.50),(104, 2000.50),(105, 1500.00); SELECT * FROM accounts; SELECT * FROM
total_bal;
```

3. Create trigger delete_orders and procedure delete_orders_orderlines to delete order. The add_order_line procedure will check the entry of lines for an order that does not exist:

```
// Create db delete_orders with tables orders and orderlines drop database if exists
delete_orders; create database delete_orders; use
delete_orders; drop table if there are
orders? create table
orders (orderno int auto_increment not
null, custno int not null, odate datetime not null, primary key(orderno)); drop table if exists orderlines? create table orderlines
(orderno int not null, stockno int not null, qty int not null,
primary key (orderno,stockno));


DESCRIBE orders?
DESCRIBE orderlines?

SET AUTOCOMMIT=0;
insert into orders (custno,odate) values (1,current_timestamp); insert into orderlines
(orderno,stockno,qty) values (1,10,1),(1,50,2); commit; select * from orders; select * from orderlines;




// Create trigger delete_orders to check deletes drop trigger IF EXISTS delete_orders;

CREATE TRIGGER delete_orders
AFTER DELETE ON orders
FOR EACH ROW
DELETE FROM orderlines WHERE orderno= OLD.orderno; /* testing */
delete from orders
where orderno=1; select * from orders; select *
from orderlines; rollback?
select * from orders; select * from
orderlines;




// Create procedure delete_orders_orderlines to check for deletes in // orderlines table

DROP PROCEDURE delete_orders_orderlines;
DELIMITER $
CREATE PROCEDURE delete_orders_orderlines(IN del_order INT)
BEGIN
    DELETE FROM orderlines WHERE orderno=del_order;
    DELETE FROM orders WHERE orderno=del_order;
END?
$
Delimiter ;
```

Machine Translated by Google

```
// Tests
Set autocommit=0;
select * from orders;
select * from orderlines;
CALL delete_orders_orderlines(1);
select * from orders;
select * from orderlines;
rollback?
select * from orders;
select * from orderlines;

// Create procedure add_order_line to check the existence of an order
DELIMITER $
CREATE PROCEDURE add_order_line(IN o_no int, IN s_no int, IN qty int)
BEGIN
DECLARE count_var INT;
SELECT COUNT(orderno)
INTO count_var
FROM orders
WHERE orderno=o_no;
IF (count_var <>1) THEN
SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT='Order number is not found in orders table';
END IF?
END?
$
DELIMITER ;

// Tests
SELECT * FROM Orders;
SELECT * FROM Orderlines;
CALL add_order_line(2,10,1);
ERROR 1644(45000): Order number is not found in orders table

CALL add_order_line(2,50,2);
ERROR 1644(45000): Order number is not found in orders table

SELECT * FROM Orders;
SELECT * FROM Orderlines;
```

4. The commit and rollback statements are not allowed in stored functions. Here is an example:

```
// Create function myFun with commit and rollback
DROP FUNCTION IF EXISTS myFun?
DELIMITER !
CREATE FUNCTION myFun (p_no INT, p_in VARCHAR(30))
RETURNS VARCHAR(30)
LANGUAGE SQL
BEGIN
INSERT INTO myTrace (t_no, t_user, t_date, t_time, t_proc, t_what)
   VALUES (p_no, current_user, current_date, current_time, 'myProc', p_in);
IF (MOD(p_no,2)=0) THEN
     COMMIT;
ELSE ROLLBACK?
END IF?
RETURN p_in;
END!
DELIMITER ;

The function is not created and returns an error:
ERROR 1422(HY000): Explicit or implicit commit is not allowed in stored function or trigger
```

5. Example of managing division by 0:

```
// Create procedure Divide_by_zero to check for division by 0
DELIMITER $$
CREATE PROCEDURE Divide_by_zero (IN numerator INT, IN denominator INT,
        OUT results INT)
BEGIN
DECLARE div_by_zero CONDITION FOR SQLSTATE '22012';
DECLARE CONTINUE HANDLER FOR div_by_zero RESIGNAL SET MESSAGE_TEXT ='Division by zero';
IF denominator = 0 THEN
SIGNAL div_by_zero;
ELSE
SET results := numerator/denominator;
END IF?
END?
$$
DELIMITER ; //
Test
CALL Divide_by_zero(100, 0, @results);
ERROR 1644(22012): Division by zero
```

6. Example of handling division by 0:

```
// Create db my_first_triggers_db with table new_dept
USE my_first_triggers_db; drop
table if exists new_dept; create table
new_dept(deptno int not null, dname char(30),
PRIMARY KEY(deptno));

// Create trigger trg_signals_raising_error to check for record inserts in // table new_dept with negative department code drop trigger if exists
trg_signals_raising_error; delimiter // create trigger trg_signals_raising_error
before insert on new_dept for each row begin declare msg varchar(255); if
new.deptno < 0
then set msg = concat('Trying to insert a negative value in deptno:
', cast(new.deptno as char)); signal
sqlstate '45000' set

message_text = msg; endif? end //
delimiter ;

// Tests insert
into new_dept values (1, 'SALES'), (-1, 'ACCOUNTING'), (2, 'RESEARCH'); Error message and global
input rejection

insert into new_dept values (1, 'SALES'); select * from
new_dept;

insert into new_dept values (-1, 'ACCOUNTING'); Error message
and input rejected
```