CS2102
Structured Query Language (SQL)
Part 3

Conditional Expressions: CASE

Scores

name	marks
Alice	92
Bob	63
Carol	58
Dave	47

name	grade
Alice	A
Bob	B
Carol	C
Dave	D

```
select name, case
        when marks >= 70 then 'A'
        when marks >= 60 then 'B'
        when marks >= 50 then 'C'
        else 'D'
end as grade
from   Scores;
```

Conditional Expressions: CASE (cont.)

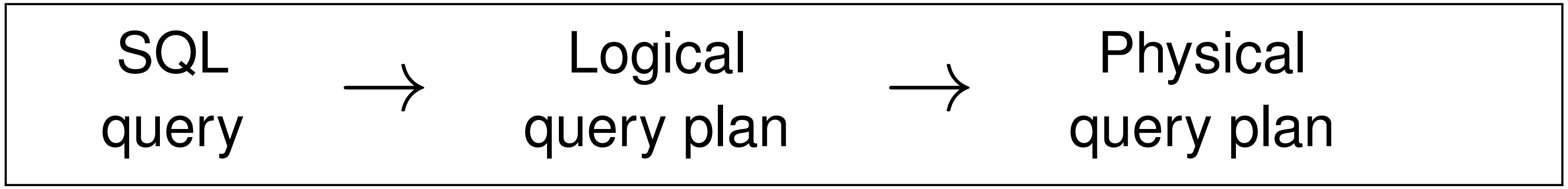
```
case
  when  $condition_1$  then $result_1$
  . . .
  when  $condition_n$  then $result_n$
  else $result_0$
end
```

```
case expression
  when  $value_1$  then $result_1$
  . . .
  when  $value_n$  then $result_n$
  else $result_0$
end
```

Other conditional expressions: **coalesce** `and` **nullif** `functions` (not covered)

SQL Query Processing

1. Query parsing & authorization
2. Query optimization



SQL query → Logical query plan → Physical query plan

3. Query execution

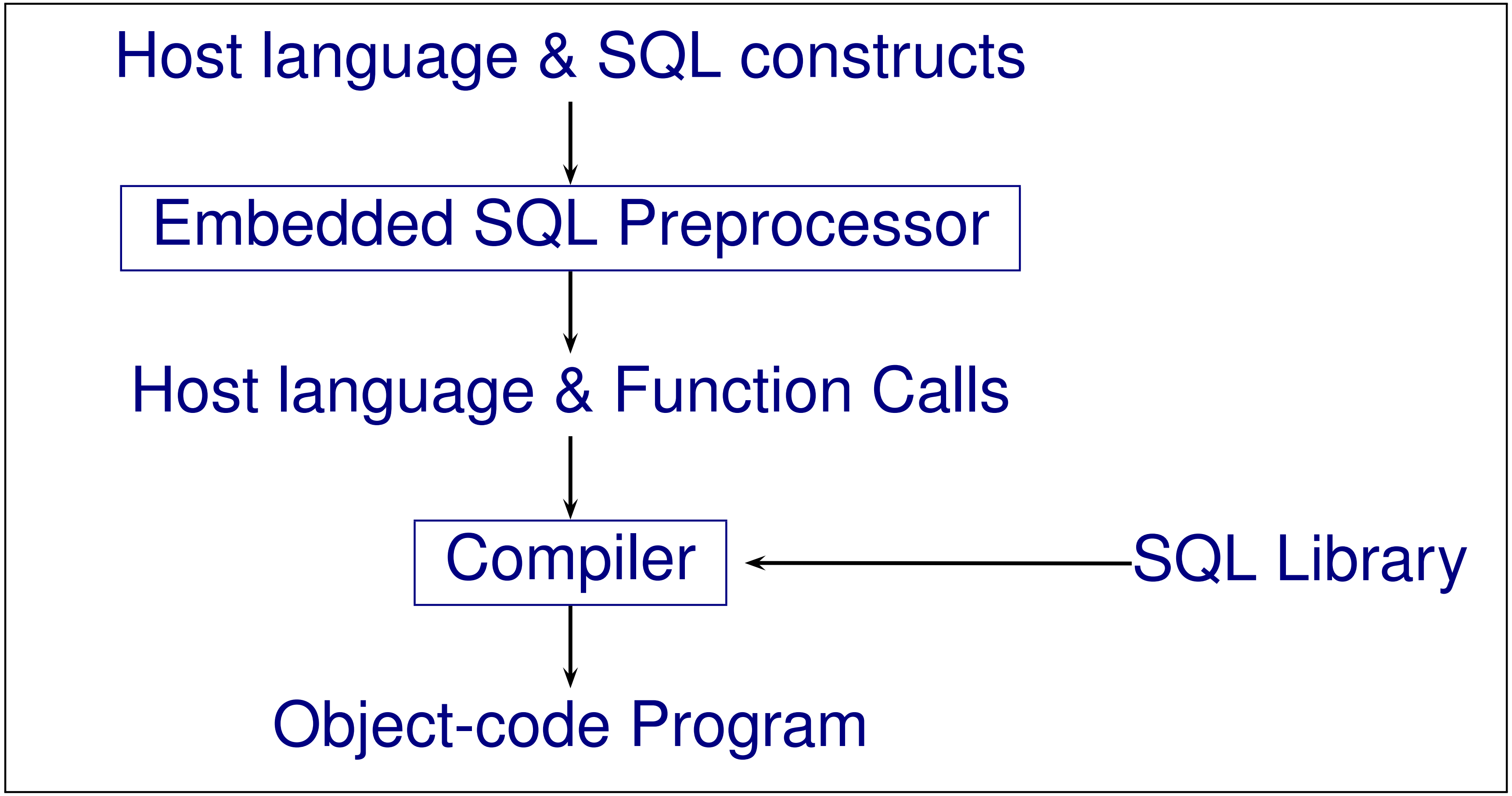
Using SQL

- **Directly write SQL statements**
  - Command line interface
    - Oracle's SQL*Plus
    - PostgreSQL's psql
    - etc.
  - Graphical interface
    - Oracle SQL Developer
    - PostgreSQL's pgAdmin
    - etc.
- **Include SQL in application programs**
  - Statement-Level Interface (SLI)
  - Call-Level Interface (CLI)

Statement-Level Interface (SLI)

- Application program combines host language & SQL constructs
- Two forms of SQL constructs:
    - Embedded SQL (a.k.a. static SQL)
    - Dynamic SQL
- Embedded SQL
    - SQL constructs are SQL statements
    - SQL statements are known at compile time
- Dynamic SQL
    - SQL constructs are directives for preparing/executing SQL statements
    - SQL statements are stored in string variables
    - SQL statements may not be known at compile time

SLI: Compiling Programs


Host language & SQL constructs
Embedded SQL Preprocessor
Host language & Function Calls
Compiler
SQL Library
Object-code Program

Embedded SQL: Example

```
int main() {
    EXEC SQL BEGIN DECLARE SECTION;
        int stuId;  char name[30]; char major[10];
    EXEC SQL END DECLARE SECTION;
    EXEC SQL CONNECT TO testdb;
    EXEC SQL WHENEVER SQLERROR GOTO query_error;
    EXEC SQL WHENEVER NOT FOUND GOTO bad_student;

    printf ("Enter student number: ");    scanf("%d", &stuId);

    EXEC SQL SELECT name, major  INTO :name, :major
        FROM Students  WHERE stuId = :stuId;

    printf ("StuId: %d  Name: %s Major: %d\n", stuId, name, major);

    EXEC SQL DISCONNECT ALL;   return 0;

query_error:
    printf ("SQL error: %ld\n", sqlca->sqlcode);   exit();
bad_number:
    printf ("Invalid student number.\n");    exit();
}
```

Embedded SQL: Cursors

```
EXEC SQL BEGIN DECLARE SECTION;
     int stuId;
     char name[30];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE stuCursor CURSOR FOR
     SELECT stuId, name FROM Students;
EXEC SQL OPEN stuCursor;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
     EXEC SQL FETCH FROM stuCursor INTO :stuId :name;
     printf("student Id = %d, name = %s\n", studId, name);
}
EXEC SQL CLOSE stuCursor;
```

Details of PostgreSQL's Embedded SQL in C:

https://www.postgresql.org/docs/current/static/ecpg.html

Dynamic SQL

```
EXEC SQL BEGIN DECLARE SECTION;
    const char *stmt = "SELECT name FROM Students WHERE stuId = ?";
    char name[30];
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
EXEC SQL EXECUTE mystmt INTO :name USING 1234567;
```

Call-Level Interface (CLI)

- Vendor-independent API for database access
- Unlike SLI, programs are written entirely in host language
- Uses string variables to construct SQL statements (similar to dynamic SQL)
- Examples of CLI APIs:
  - JDBC (Java DataBase Connectivity)
    - `https://jdbc.postgresql.org/`
  - ODBC (Open DataBase Connectivity)
    - `https://odbc.postgresql.org/`

JDBC: Example

```
import java.sql.*;
.....
String      url, userId, password;
Connection  conn;
.....

try {
    Class.forName("org.postgresql.Driver");
    conn = DriverManager.getConnection(url, userId,
        password);
} catch (ClassNotFoundException e) {
    System.err.println("Can't load driver\n");
    System.exit(1);
} catch (SQLException e) {
    System.err.println("Can't connect\n");
    System.exit(1);
}

```

JDBC: Example (cont.)

```java

int               year = 4;
PreparedStatement   stat = conn.prepareStatement("SELECT * FROM Students WHERE year = ?");
stat.setInt(1, year);
ResultSet rset = stat.executeQuery();

while (rset.next())
{
    System.out.print("Column 1 value");
    System.out.print(rs.getString(1));
}

rset.close();
stat.close();
conn.close();
```

SQL Injection Attacks

- SQL injection attacks are a type of injection attack
- **Source**: `https://www.owasp.org/index.php/Top_10_2007-Injection_Flaws`
  Injection occurs when user-supplied data is sent to an interpreter as part of a command or query. Attackers trick the interpreter into executing unintended commands via supplying specially crafted data. Injection flaws allow attackers to create, read, update, or delete any arbitrary data available to the application. In the worst case scenario, these flaws allow an attacker to completely compromise the application and the underlying systems, even bypassing deeply nested firewalled environments.

SQL Injection Attacks (cont.)

- Consider the following dynamic SQL query:

  "SELECT * FROM R WHERE x =’" + var + "’"

  where `var` is a variable for capturing user’s input

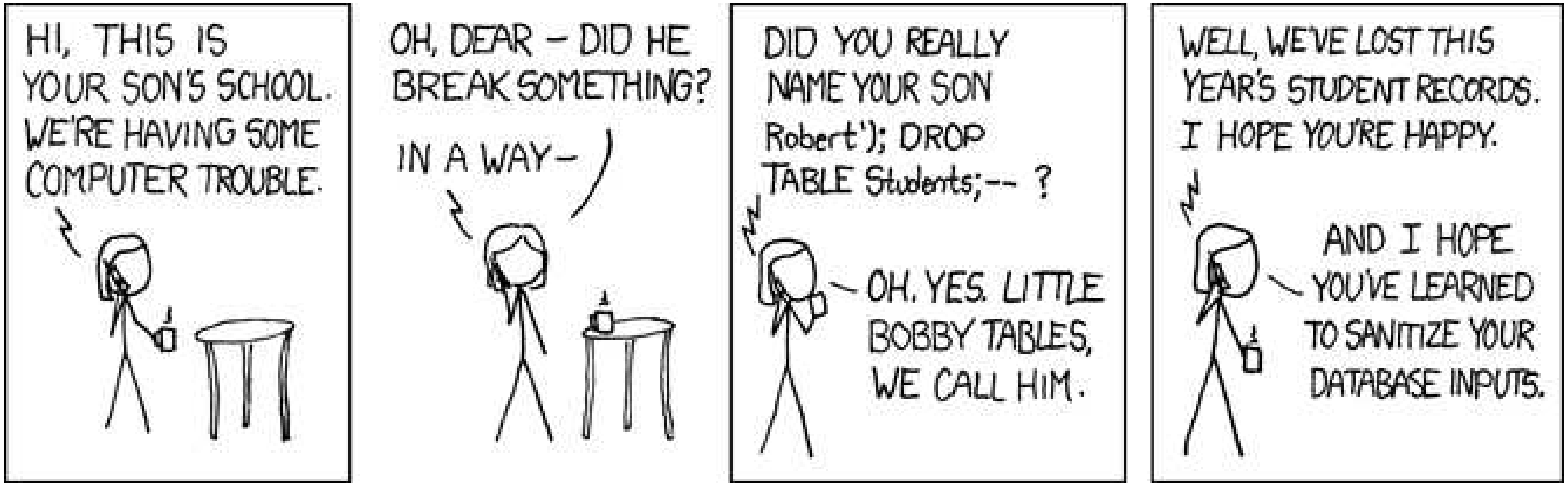
- If the user inputs the following value for `var`:

  0’ or ’1’ = ’1

  the query becomes

  SELECT * FROM R WHERE x =’0’ or ’1’ = ’1’

SQL Injection Attacks (cont.)


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Source: `https://xkcd.com/327`

SELECT * FROM Students WHERE name ='Robert'; DROP TABLE Students; - -'

SQL Injection Attacks: Prevention

- Use prepared statements - appropriate escape characters will be inserted into input string

SELECT * FROM R WHERE x =’0’ or ’1’ = ’1’

↓

SELECT * FROM R WHERE x =’0\’ or \’1\’ = \’1’

SQL Injection Reference:

`https://www.owasp.org/index.php/SQL_Injection`

ANSI SQL Isolation Levels

- The isolation level for a transaction affects what the transaction will read
- ANSI SQL defines four isolation levels
  - Read Uncommitted (weakest isolation level)
  - Read Committed
  - Repeatable Read
  - Serializable (strongest isolation level)
- Choice of isolation level affects correctness vs performance tradeoff
- In many DBMSs, the default isolation level is Read Commited
- Configure using **set transaction isolation level** statement

Serializable Transaction Executions

- Consider a set of transactions $S = \{T_1, \cdots, T_n\}$
- An execution of S is a serial execution if the execution of the transactions in S are not interleaved
- An execution of S is serializable if it is equivalent to some serial execution of S
- **Serializable executions** guarantee correctness of transaction executions

Transaction: Example

```
int Transfer (int fromAcctId, int toAcctId, int amount)
{
    EXEC SQL BEGIN DECLARE SECTION;
        int fromBalance;    int toBalance;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL WHENEVER SQLERROR GOTO query_error;

    EXEC SQL SELECT balance INTO :fromBalance FROM Accounts
        WHERE accountId = :fromAcctId;
    if (fromBalance < amount) {
        EXEC SQL ROLLBACK;   return 1;
    }
    EXEC SQL SELECT balance INTO :toBalance FROM Accounts
        WHERE accountId = :toAcctId;
    EXEC SQL UPDATE Accounts SET balance = :toBalance + :amount
        WHERE accountId = :toAcctId;
    EXEC SQL UPDATE Accounts SET balance = :fromBalance - :amount
        WHERE accountId = :fromAcctId;
    EXEC SQL COMMIT;
    return 0;
    query_error: printf ("SQL error: %ld\n", sqlca->sqlcode); exit();
}
```

Serial Transaction Executions

Consider the executions of Transfer(1,2,100) & Transfer(2,1,100)

```
begin transaction;
select      balance into :xbal
from        Accounts
where       accountId = 1;
select      balance into :ybal
from        Accounts
where       accountId = 2;
update      Accounts
set         balance = :ybal + 100
where       accountId = 2;
update      Accounts
set         balance = :xbal - 100
where       accountId = 1;
commit;
begin transaction;
select      balance into :ybal2
from        Accounts
where       accountId = 2;
select      balance into :xbal2
from        Accounts
where       accountId = 1;
update      Accounts
set         balance = :xbal2 + 100
where       accountId = 1;
update      Accounts
set         balance = :ybal2 - 100
where       accountId = 2;
commit;
```

```
begin transaction;
select      balance into :ybal2
from        Accounts
where       accountId = 2;
select      balance into :xbal2
from        Accounts
where       accountId = 1;
update      Accounts
set         balance = :xbal2 + 100
where       accountId = 1;
update      Accounts
set         balance = :ybal2 - 100
where       accountId = 2;
commit;
begin transaction;
select      balance into :xbal
from        Accounts
where       accountId = 1;
select      balance into :ybal
from        Accounts
where       accountId = 2;
update      Accounts
set         balance = :ybal + 100
where       accountId = 2;
update      Accounts
set         balance = :xbal - 100
where       accountId = 1;
commit;
```

Non-Serializable Execution: Example

Consider the executions of Transfer(1,2,100) & Transfer(2,1,100)

```
begin transaction;
select     balance into :xbal
from       Accounts
where      accountId = 1;
select     balance into :ybal
from       Accounts
where      accountId = 2;
update     Accounts
set        balance = :ybal + 100
where      accountId = 2;
                                        begin transaction;
                                        select     balance into :ybal2
                                        from       Accounts
                                        where      accountId = 2;
                                        select     balance into :xbal2
                                        from       Accounts
                                        where      accountId = 1;
update     Accounts
set        balance = :xbal - 100
where      accountId = 1;
commit;
                                       update     Accounts
                                       set        balance = :xbal2 + 100
                                       where      accountId = 1;
                                       update     Accounts
                                       set        balance = :ybal2 - 100
                                       where      accountId = 2;
                                       commit;
```

Serializable Execution: Example

Consider the executions of Transfer(1,2,100) & Transfer(2,1,100)

```
begin transaction;
select      balance into :xbal
from        Accounts
where       accountId = 1;
select      balance into :ybal
from        Accounts
where       accountId = 2;
update      Accounts
set         balance = :ybal + 100
where       accountId = 2;
                                           begin transaction;
                                           select      balance into :ybal2
                                           from        Accounts
                                           where       accountId = 2;
update       Accounts
set          balance = :xbal - 100
where        accountId = 1;
commit;
                                         select      balance into :xbal2
                                         from        Accounts
                                         where       accountId = 1;
                                         update      Accounts
                                         set         balance = :xbal2 + 100
                                         where       accountId = 1;
                                         update      Accounts
                                         set         balance = :ybal2 - 100
                                         where       accountId = 2;
                                         commit;
```

Summary

- Programming with SQL
  - Statement-level Interface (SLI)
  - Call-level Interface (CLI)
- SQL Injection Attacks
  - Use prepared statements to prevent attacks
- SQL isolation levels
  - Serializable transaction executions guarantee correctness
  - **set transaction isolation level serializable**;