EasyQL
Language Reference Manual

CS W4115: Programming Languages and Translators
Professor Stephen A. Edwards
Computer Science Department
Fall 2006 Columbia University

EasyQL Members:

Kangkook Jee (kj2181)
Kishan Iyer (ki2147)
Smridh Thapar (st2385)
Sahil Peerbhoy (sap2126)

Table of Contents

1. Lexical Conventions

A program consists of operations that the user wants to perform in different databases managed by different database management systems. Programs are written using the ASCII character set with some combination of characters reserved as constants.

1.1. Comments

Both C- and C++-style comments are supported. A C-style comment begins with the characters /* and ends with the characters */. Any sequence of characters may appear inside of a C-style comment except the string '*/'. C-style comments do not nest. A C++-style comment begins with the characters // and ends with a line terminator. /* and */ have no special meaning inside comments beginning with //. // has no special meaning inside comments beginning with /*.

1.2. Tokens

A token is the smallest element that is meaningful to the compiler. The EasyQL parser recognizes these kinds of tokens as: identifiers, keywords, literals and operators. A stream of these tokens makes up a translation unit.
Tokens are usually separated by white-space (for details, see 1.5.2 Character constants). White space can be one or more:

- Blank spaces
- Horizontal or vertical tabs
- New lines
- Comments

1.3. Identifiers

An identifier is a sequence of alphabets or digits or '_' but it cannot start with a digit. There is no limit on the length of an identifier. Two identifiers are the same if they have the same ASCII code for every character.

Identifier: *(letter|'_')(letter | digit | '_')**

1.4. Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

varchar int float date table connection
if else while display connection

1.5. Literals

Invariant program elements are called literals or constants. The terms literal and constant are used interchangeably here. Literals fall into four major categories: integer, character, floating-point, and string literals.

1.5.1. Integer Constants

Integer constants are constant data elements that have no fractional parts or exponents. They always begin with a digit. You can specify integer constants in decimal, octal, or hexadecimal form. They can specify signed or unsigned types and long or short types.

1.5.2. Character Constants

Character constants are one or more members of the source character set, the character set in which a program is written, surrounded by single quotation marks ('). They are used to represent characters in the execution character set, the character set on the machine where the program executes.

Reserved Characters:

1	New line	\n
2	Horizontal tab	\t
3	Vertical tab	\v
4	BEL	\\
5	Question mark	\?
6	Single quotation mark	\'
7	Double quotation mark	\"
8	Null character	\0

1.5.3. Floating Point Constants

Real constants specify values that may have a fractional part. These values contain decimal points (.) and can contain exponents.

1.5.4. String Constants

A string literal consists of zero or more characters from the source character set surrounded by double quotation marks ("). A string literal represents a sequence of characters that, taken together, form a null-terminated string.

1.6. Embedded SQL Blocks

Embedded SQL blocks begin with the character sequence '<$' and end with the character sequence '$>'. The embedded code will be inserted into the generated query set. The compiler may inspect the SQL code and issue errors or warnings, but it is not required to do so.

SQL block: *<$ SQL statement $>*

2. Data Types

EasyQL uses data types which are commonly used and are compatible with a large number of databases. This is done to ensure portability of tables defined in EasyQL. The categories of the data types are:

- Character data types
- Number data types
- Date data type
- Database object data types

2.1. Character Types

2.1.1. Characters – type 'varchar'

Like the "varchar" type of SQL, this creates a character string of variable length, the maximum length in bytes restricted by "size". Unlike the char type, there is no blank padding when fewer than "size" characters are supplied. However, inserting a value greater than "size" characters is not permitted. Their declaration syntax is:

varchar(size) identifier_name ;

2.2. Number Types

2.2.1. Integer – type 'int'

It maps on to the SQL type "INTEGER" and represents a 32-bit (signed) integer value. The value will range between integers -2147483648 and 2147483647. Their declaration syntax is:

int identifier_name;

2.2.2. Real – type 'float'

This type corresponds to the SQL type "REAL". In EasyQL, we define the float type to contain 15 bits of mantissa. Their declaration syntax is:

float identifier_name;

2.3. Date Type

2.3.1. Date – type 'date'

The date type in EasyQL represents a date comprising a date, month and year as well as for time. This will correspond to "DATE" in SQL. Their declaration syntax is:

date identifier_name ;

2.4. Database object types

2.4.1. Table – type 'table'

The table type in EasyQL is a collection of the attributes of a table. The attributes may be of different types and attributes within a table must have unique names, in order to distinguish between them. Their declaration syntax is:

table identifier;

Table variable can be initialized by retrieving existing table in the connection or by creating new one.
An attribute *name* of a table *tableA* could be accessed thus*: tableA[id]*

2.4.2. Connection – type 'connection'

The connection type specifies a connection to a particular database. A connection instance can be created by passing the right connection parameters when creating a connection type. Their declaration syntax is:

connection identifier;

Assignment to connection can be made by calling connect () built-in function

Ex. connection conn = connect (localhost, 1521, testDB, mysql, msyqlpasswd);

Since the connection is to a particular database, an attribute *id* of table *table1* in a databsase connected to *conn* can be assigned to a variable *attr1* by:

Ex. attr1 = conn:table1[id]

3. Expressions and Operators

An Expression is any sequence of operators and operand which produces a value or generates a side effect. We'll define expressions and operators of every data types in our language. Firstly, we'll start with discussing simple expressions that are called as primary expressions

3.1. Primary expressions

This represents simple expressions. It includes previously declared identifiers, numbers, characters, tables, connections and expressions.

primary-expression: *identifier-expression*
| *number-expression*
| *date-expression*
| *character-expression*
| *table-expression*
| *connection-expression*
| *expression*

3.1.1. Identifier

An identifier is a primary expression provided it is declared as designating an object.

3.1.2. Number

A number is a primary expression. Its type depends on its form (integer or float) see 1.5.1 Integer Constants and 1.5.3 Real Constants.

3.1.3. Character

A character is a primary expression. See 1.5.2 Character Constants.

3.1.4. Date

A date is a primary expression. It represents year, month, date, time, minute and second. See 1.5.3 Date constants.

3.1.5. Table

A table is a primary expression. See 2.4.1 Table type.

3.1.6. Connection

A connection is a primary expression. See 2.4.2 Connection type.

3.2. Expression and parenthesized-expression

Any expression can be regarded as a primary expression. And an expression within parentheses has the same type and values as the expression without parentheses would have. Any expression can be delimited by parentheses to change the precedence of its operators.

expression: (expression)

3.2.1. Expressions for basic data types

This part is going to cover operators and expressions of basic types of EasyQL. Which are character, number and date data types. Other than those, the language contains database object data types and these will be mentioned in the later part of this manual. The precedence of expression operators is the same as the order of the major subsections of this section (the highest precedence first). The rules of association for operators are mentioned in each section.

3.2.1.1. Postfix expressions

These are two postfix operators for incrementing and decrementing objects for number types. These will be placed after the operand and change the value of operand. The type won't be changed after operation. Number type of expressions can be operand for these expressions. These are right-to-left operation.

postfix-expression: number-expression ++
| number-expression --

3.2.1.2. Unary operations + , - , ++, --,

These are unary operators that will be placed in front of operand and change the value of operand. The type won't be changed after operation. Number type of expressions can be operand for these expressions. These are left-to-right operations.

prefix-expression: + number-expression
| - number-expression

prefix-expression: ++ number-expression

prefix-expression: -- number-expression

prefix-expression: !number-expression

3.2.1.3. Binary operations + - * / %

Theses are binary operations and which means addition, subtraction, multiplication, division and modular operation respectively. Number type of expression can be operands for these expressions and these will return the same type expressions. These are left-to-right operations.

*binary-expression: number-expression * number-expression*

binary-expression: number-expression / number-expression

binary-expression: number-expression % number-expression

binary-expression: number-expression + number-expression

binary-expression: number-expression - number-expression

3.2.1.4. Comparison operators == != >= > < <= , like

These operators will compare operands and evaluate relationship between operands. These will return 0 if the relationship in the expression doesn't hold, 1 otherwise. Operand for this operation can be number type expression, and date type expressions. These operations are for left-to-right evaluation.

compare-expression: compare-expression == compare-expression

compare-expression: compare-expression != compare-expression

compare-expression: compare-expression > compare-expression

compare-expression: compare-expression >= compare-expression

compare-expression: compare-expression < compare-expression

compare-expression: compare-expression <= compare-expression

3.2.1.5. Logical operators && ||

These operators will yield 1 (in case of TRUE) and 0 (in case of FALSE) otherwise. These operations are for left-to-right evaluation.
This will return 1 if both operands are non-zero, 0 otherwise.

logical-expression: logical-expression && logical-expression

This will return 1 if any of operands is non-zero, 0 otherwise.

logical-expression: logical-expression || logical-expression

3.2.1.6. Assignment expression

The language provides one assignment operator as bellows. This will be used to assign a value of right expression to the modifiable identifiers of the left.

assignment-operation: identifier-expression = expression

3.2.2. Operators for table type and connection type

In this section, we will cover operations between database objects. We will mostly deal with the operations of table data types and a couple of operators for connection data types will be mentioned briefly.

3.2.2.1. Sub-Table Operators [ ], ()

Programmers can retrieve data from a table type variable with these operators. These operators are to specify appropriate attributes and conditions. The output of sub-table operation is originally meant to be another type of table object. But in case of having just one element as a result of query, this can be treated as basic data types such as numbers, characters and date data types.

sub-table-expression: *table-expression [attr-expression-list]*
sub-table-expression: *table-expression (cond-expression-list)*

attr-expression-list(opt): *attr-expression, attr-expression-list(opt)*
cond-expression-list: *cond-expression, cond-expression-list*

Ex. ***tableA[ssn, name].display()***

This example operation will retrieve sub-table of ssn attribute and name attribute from *tableA*. *display ()* function will print the records in sub-table.

Ex. ***tableA(age>20).display();***

This example operation will retrieve sub-table of tableA which satisfies the condition that is specified in the parenthesis. *display ()* function will print the records in sub-table

Ex. ***tableA[name](age>20).display();***

Programmer can specify attributes and conditions at the same time.

Ex. ***table tableB;***
tableB =tableA.[name](age>20).display();

Result of functions can be assigned to another table type variable, instead of displaying output to standard output.

Ex. ***table tableC***
int average_age;
average_age = tableA[avg(age)];

Programmers can make use of group functions already defined in SQL language. These are *avg(), count(), max(), min(),* and *sum()*. Since the output value of this expression is just a sub-table of one attribute and one record. This can be directly assigned to integer data type.

3.2.2.2. Table Update Expression

This operation is to update and changing the existing value of the table.

table-update-expression: table-expression.update [assignment-expression-list] (cond-expression-list)

assigment-expression-list: assignment-expression, assignment-expression-list (opt)
cond-expression-list: cond-expression, cond-expression-list

Ex. ***tableA.update [cuid="123456"](name=="Micheal")***

This will update cuid attribute of a record where the value of name attribute is "Micheal".

3.2.2.3. Table Insert Expression

This operation is to insert a record into a table.

insert-expression: table-expression.insert (attribute-list)
table-expression.insert (table-expression)

attribute-list: attr-name:=attr-value, attribute-list

Ex. ***tableA.insert (name="Micheal", age=20, cuid=123456);***

Operation in above example expression will insert a record into a existing table.

3.2.2.4. Table Delete Expression

This operation is to delete a record from a table.

delete-expression: table-expression.delete (cond-expression-list)

cond-expression-list: cond-expression, cond-expression-list

Ex. ***tableA.delete (name=="Micheal");***

Operation in above example expression will delete a record from a table.

3.2.2.5. Create Table Expression

This operation is to create a table according to the schema that specified by a programmer.

create-expression: table-name-expression.create (attr-decl-list)

attr-decl-list: type-expression id-expression, attr-decl-list

Ex. ***table tableA;***
tableA.create (varchar name, int age, int cuid);

3.2.2.6. Drop Table Expression

This operation will cancel definition of table type object and drop the table schema in the table space.

drop-expression: table-name-expression.drop()

Ex. ***tableA.drop();***

This will drop the existing table from the table space

3.2.2.7. Metadata Expression

This operation *.desc()* and *.list()* can work on table or connection type and will retrieve the schema of the table and tables and connection related information (such as user name, address of database, database instance name …) that belongs to the connection respectively. Output of operations can be stored as table data-type or printed to the user screen by calling a built-in function *.display ()*.

meta-expression: table_expression.desc ();
| connection_expression.desc();

Ex. ***table tableB tableA.desc();***
tableB.display();

The next operation *.list()* can be applied only to connection data type to list table names which belongs to the connection.

meta-expression:connection_expression.list();

Ex. ***connection conn = connect (localhost,1521,testDB, mysql,mysql123);***
conn.list().display();

3.2.3. Operators for Connection Type

The *"::"* operation will retrieve a table data type from a connection.

connection-expression: connection-expressio::.table-name

Ex. ***connection A = connect (localhost,1541,tablespaceA, mysql,passwd)***
table tableA = A::ProjectMemberTable;

The *.store* operation will store a table data type into a connection.

connection-expression: connection-expression.store(table-name);

Ex. ***connection A = connect (localhost,1541,tablespaceA, mysql,passwd)***
connection B = connect (remotehost,1541,tablespaceB,mysql,passwd)
table tableA = A::ProjectMemberTable;
B.store(A);

4. Statements

Statements are generally always executed is sequence.

4.1. Expression statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

4.2. Compound statement

So that several statements can be used where one is expected, the compound statement is provided

compoundstatement:
{
statementlist
}
statementlist: statement statement statementlist

4.3. Conditional Statement

The two forms of the conditional statement are

if (expression) statement
if (expression) statement else statement

In both cases the expression is evaluated and if it is nonzero, the first sub-statement is executed. In the second case the second sub-statement is executed if the expression is 0. As usual the ‘‘else’’ ambiguity is resolved by connecting an else with the last encountered else-less if.

4.4. While statement

The while statement has the form

while (expression) statement

The sub-statement is executed repeatedly so long as the value of the expression remains nonzero. The test takes place before each execution of the statement.

4.5. PRE – DEFINED FUNCTIONS

4.5.1. Display

This is a predefined function that is used to display the results of a query on the screen.

database-expressions.display()

4.5.2. Connection

This is a predefined function that creates the connection to the database.

connecton-type connection (ipaddress , port, instance_name, db_name, user, password);

5. Examples

This section deals with some exemplary codes that can illustrates usages of EasySQL

5.1. Table insertion example

This program is to merge a table from one database to the table of the other.

connection A = connect(localhost, 1541, tablespaceA, root, passwd);
connection B = connect(remotehost, 1541, tablespaceA, root, passwd);

table emp1 = A::employee;
table emp2 = B::employee;

emp2.insert (emp1);

```
emp2.insert(name="john",age=30,ssn="34532198760");

emp2.display();
```

5.2. Connection retry example

In this example, the script is to make a connection to the database in the remote.
A built-in function connect () is to return a connection data type in case of success, otherwise it will return NULL.

```
int i=0;
connection b;

while(i<10||b!=NULL)
{

        b=connect(remotehost, 1541, tablespaceA, root, pwd);
        i++;
}

if(!b)
{
        display("Connection Failed");
}
```

5.3 formatted output.

This is an example of formatting output of query. In this example, script open a table from a database reside in *remotehost* and get a query output in a formatted way.

```
connection empDB = connection (remotehost, 1541, tablespace, root,passwd);
table emp = empDB:employee;

empDB["Mr."+last_name] (sex=="male").display();
empDB["Miss."+last_name] (sex=="female").display();
```