EasyQL
A Data Manipulation Language

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. Whitepaper

Introduction

The EasyQL language is a database manipulation language where the primary aim is to allow users to handle and manipulate data from several database instances simultaneously. The language will simplify for the programmer, tasks such as connecting to a database and performing operations on data residing on different database instances.

Why EasyQL?

We believe that while traditional database-related tasks such as querying a table, inserting/deleting records and so on still make up the majority of the operations on data, there is still a need to be able to handle various types of data within a single environment. This is what EasyQL seeks to address.

What are we going to implement?

We are trying to implement a simple interpretive environment from which a user can connect to multiple database instances at a time and can easily switch from one connection to another. Manipulation and transfer of data between tables, tablespaces and instances will be enabled.

While SQL is meant to operate on data from a table, between tables and at most between tablespaces, EasyQL focuses on a broader scope of data objects such as ones from different connections, different instances and various metadata that define the attributes of a database.

EasyQL will follow a syntax similar to Java in most cases and will mirror SQL as far as operations on tables are concerned.

You may encounter the situation that you need to access retrieve data from legacy database that resides on decade's old mainframe as a form of hierarchical db or ISAM file that you never heard of or learn how to play with it. You may want to move and store your personal database (might be in MS ACCESS or anything else) into bigger DBMS (such as Oracle or MYSQL) to share it from the web.

The solution for these situations is what EasyQL will provide eventually i.e. interchange and migrate data from multiple database connections under the same interface.

Features

To better explain the features of our language, consider the following 2 tables which reside in different databases on the servers of different companies.

Table schema for Employee table from HR_tablespace of severus db instance

Id	Number (7)
Name	Varchar2 (25)
Salary	Number (11,2)
Title	Varchar2 (25)

Table schema for Employee table from HR_tablespace of test db instance

Id	Number (7)
Name	Varchar2 (25)
Salary	Number (11,2)
Title	Varchar2 (25)
Dependent	Number (2)

- **Supports multiple database connections**

EasyQL allows the programmer to connect to multiple database instances from the command line. An example showing connections being established to 3 different databases is illustrated below.

```
connection C, D;

C = conn("24.22.220.234", 3306, "test", "mysql", "root","plt123");
D = conn("10.24.168.1", 6687, "severus", "mysql", "eva", "coms4115");
```

- **Operations such as viewing table metadata**

The language enables one to compare the metadata of tables residing on different databases. If there is a match, one can perform operations such as appending rows from a table in one database instance to another. A possible append operation which our language will include is shown below.

```
table T1,T2;
T1=C::Employee;
T1.dsc();
```

Returns a description of the "projects" table residing on the host which connection C is set to.

- **Procedural and program oriented**

Enhancements with respect to most present query languages include the availability of procedural and control flow in EasyQL. For instance, upon a successful connection, one could use the following code to insert values into a database table.

```
ret = MyTable.create(C,"name varchar(255), age int");

if (ret)  {
      MyTable =C:: MyTable;
      int i1;
      i1 = 0;
      while(i1<5) {
      i1=i1+1;
      MyTable.insert["name,age"]("'Saahil',23");
      }
      display (MyTable);
} else {
      display ("create table failed");
}
```

- **SQL equivalent statements**

Creating a table requires specification of the connection on which the table is to be created.

```
table tab1;
tab1.create(D,"name varchar(20), age integer, empid varchar(10)");

tab1.insert[“name, age, empid”](“’Jenni’, 29, ‘r0987345’);

tab1.update ["name='Jennifer'"]("age=29");

tab1.delete(“age=29”);
```

Who is going to use EasyQL?

We feel that a regular database administrator whose working environment is multi-vendor oriented will need to manipulate various data from different databases at the same time will find the language rather useful. This will also be a very useful tool for those who are working in projects which involve data migration, data integration, data mining etc.

Summary

The EasyQL Language provides some flexibility for a programmer who needs to operate on data residing in different databases. The language, as the name suggests, is easy to use and should find some utility among several users.

2. Language Tutorial

The Language EasyQL is a DML (data manipulation language) that has C like syntax. It aims at providing the user the option to connect and perform operations on different databases. In this section we will explain the various constructs in the language and explain the usage of the language.

A Simple Program

```
int i;
i=0;

while (i<10)
{
      display (i);

      i=i+1;

      if (i==10) {
            display ("i is 10, exiting");
      }
}
```

The above program creates an integer ‘i’ and executes a loop to print integer from 0 to 10. Its syntax is pretty much like C with a difference that we use a “display();” function to print results.

The data types in EasyQL besides int are float, varchar, table and connection. The table and connection objects are the main data types that handle the database operations. We will explain a few examples using them.

```
connection c1;
C = conn("69.22.220.234", "3306", "test", "mysql", "root",
table MyTable;
int ret;

ret = MyTable.create(C,"name varchar(255), age int");


if (ret)  {
      MyTable =C:: MyTable;
      int i1;
      i1 = 0;
      while(i1<5) {
      i1=i1+1;
      MyTable.insert["name,age"]("'Saahil',23");
```

```
    }
    display (MyTable);
} else {
    display ("create table failed");
}
```

The above program creates a table ‘My Table’ on the connection C that is specified by the programmer. It then inserts 5 tuples in the table and then prints out the content of the table.

EasyQL allows the user to easily fetch all (or some) contents of a table into another table. We will illustrate this in the following example.

```
table T1,T2;
connection C;

C = conn("69.22.220.234", "3306", "test", "mysql", "root","plt123");

T1=C::Employees;

T2=T1["ssn, name, age"]("salary>100000");

display (T2);
```

The above program takes a table name ‘Employees’ which exists on the connection specified above it. Another table ‘T2’ is then created which is assigned to a subset of the table ‘T1’ (which contains the Employees table). The content of this table ‘T2’ (sub table from ‘T1’) is then printed. Please note, that table ‘T2’ and ‘T1’ will contain separate content. Further update on any of the two tables won’t affect the other.

Various ‘row’ operations are supported by EasyQL which provides the user the ability to get the values in the desired rows and columns of a table. These operations are enumerated below.

‘fetch (Table_Name["<COLUMN_NAME>"])’ gets the value in the mentioned column of the current row.
‘Table_Name.next()’ moves the focus to the next row of the table.
‘Table_Name.reset()’ moves the focus to top of the table.

Row level operations are also supported in EasyQL. Examples of our insert, update and delete operations are given below:

A row is inserted into the table using the following statement. The string in the square brackets contains the columns which are affected, while the string in the parentheses specify the values:

T1.insert["ssn, name, age"]("'0987785432', 'Johann', 21");

The update statement updates the row matching the condition mentioned in the parentheses and sets the values as given in the string within the brackets:

T1.update["age=22"]("name = 'Johann'");

The row matching the specified condition is deleted from the table:

T1.delete("age = 22");

The main feature of EasyQL Language is the to perform operations between different databases. It allows you to fetch table from one database and save it to another. We will illustrate this in the following program.

```
connection c1,c2;
table players;

c1 = conn("69.22.220.234", "3306", "test", "mysql", "root","plt123");
c2 = conn("localhost", "3306", "test", "mysql", "root", "plt123");

players=c1::players;

players.store(c2);
display (c2::players);
```

The above code creates two connections to separate databases. The main intention for the above code is to take a table from one database and push it to another database. The program creates a new table ‘players’ and loads the content from table ‘player’ on ‘c1’. This table is then stored on ‘c2’.

The other constructs in EasyQL are mentioned in the Language Reference Manual.

3. Language Reference Manual

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.

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 /*.

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

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 | '_')**

Keywords

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

varchar int float table connection
if else while display connection
update insert delete drop create list
dsc display store next reset fetch

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, float, and string literals.

Integer Constants

Integer constants are constant data elements that have no fractional parts or exponents. They always begin with a digit.

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

Floating Point Constants

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

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.

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
Database object data types

Character Types

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 ;

Number Types

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;

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;

Database object types

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.

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 conn () built-in function

Ex. Connection c;

C = conn(“localhost”, 6544, “test”, “mysql”, “root”, “plt123”);

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. MyTable = C ::table1

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

Primary expressions

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

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

Identifier

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

Number

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

Character

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

Table

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

Connection

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

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)

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.

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

prefix-expression: !number-expression

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

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

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

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

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.

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 cond-expression-list

attr-expression-list(opt): ["attribute-list"]
cond-expression-list: ("condition-list")

Ex. tab1 = tableA["ssn, name"](" ");)

This example operation will retrieve sub-table of ssn attribute and name attribute from tableA.

Ex. tab1 = tableA["*"](age>20);

This example operation will retrieve sub-table of tableA which satisfies the condition that is specified in the parenthesis.

```
Ex. tab1 = tableA[name](age>20);
    display(tab1);
```

Programmer can specify attributes and conditions at the same time. display () function will print the records in sub-table.

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

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 = fetch(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. The fetch() function will fetch the

Table Update Expression

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

table-update-expression:
table-name.update attr-expression-list cond-expression-list

attr-expression-list(opt): [“attribute-list”]
cond-expression-list: (“condition-list”)

Ex. tableA.update [“cuid=’123456’”](“name=’Micheal’”);

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

Table Insert Expression

This operation is to insert a record into a table.

insert-expression: table-name.insert attr-expression-list cond-expression-list

Ex. tableA.insert [“name, age, cuid](“’Abraham’, 34, ‘C0078676’”);

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

Table Delete Expression

This operation is to delete a record from a table.

delete-expression: table-name.delete cond-expression-list

Ex. tableA.delete (“name=’Micheal’”);

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

Create Table Expression

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

create-expression: table-name.create (connection-name, “attribute-list”);

Ex. table tableA;
tableA.create (con1, "name varchar, age integer");

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.drop()

Ex. tableA.drop();

This will drop the existing table from the table space.

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.dsc ();

Ex. 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;
connectMe =
conn ("localhost",1521,"testDB"," mysql","mysq","l123");
connectMe.list();

Operators for Connection Type

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

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

Ex. connectioin A ;
A = conn (localhost,1541,tablespaceA., root,passwd);
table tableA = A::ProjectMemberTable;

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

table-name.store(connection-name);

Ex. tableA.store(A);

Statements

Statements are generally always executed is sequence.

Expression statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

Compound statement

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

compoundstatement:
{
statementlist
}
statementlist: statement statementlist

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.

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.

PRE – DEFINED FUNCTIONS

Display

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

display(EQDataType);

Connection

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

Connecton connection-name;
Connection-name = conn (“ipaddress” , port,” instance-name”, “db-name”,” user”, “password”);

4. Project Plan

Planning

To make sure our progress did not stagnate and to ensure regular progress, our team would meet at least once a week. Every team member would update the others on the work they accomplished during the week and/or any problems they faced. We also made it a point to regularly contact our TA, Deepti Gupta, and update her with our progress.

Communication was the key to keeping our project on track. We found that since most of us stayed far apart e-mail was the best way to keep tack of progress. Many important design decisions were made via e-mail if a meeting was not possible.

To oversee meetings and guide our course of action, Kangkook Jee was made the team leader. Before every meeting we would set an agenda and Kangkook would bring us back to the focal point in case we strayed. Another helpful task was mailing the outcome of every meeting to every group member. We would take turns at this.

Development

We spent a major part of the semester designing our language. We intended to keep the basic syntax close to C/Java so that user's would have less trouble picking it up. The database specific part for our language took more discussion as it is the crux of our language. We had to make sure that our language would make it easy for user's to perform certain database specific functions that we felt were lacking in other languages.

Before we started the full fledged process of coding we each set up CVS on our individual systems. This way we could each make updates to any module and avoid making redundant and conflicting updates. It was impressed upon every teem member to commit every change they made, no matter how small.

Our first milestone was completing the lexer and the parser. This was the foundation on which the rest of the code could be created. Once we had a functional lexer and parser we started building the tree walker and testing our code in parallel.

The work was generally divided between groups of two people. This way our progress would not be limited by any one individuals schedule and progress would be fast.

Testing

We followed an incremental strategy to produce quality code. Rather than code an entire module and then fix bugs, we would check each new functionality as it was added. This helped to isolate bugs and speeded up the process of testing.

Special test cases were created by Smridh which were sample programs written in our language and were used to make sure that no new major bugs were introduces whenever major changes to the lexer/parser and the tree walker were made.

Team Contribution

Kangkook Jee	Lexer, Parser, Tree Walker and Back End
Saahil Peerbhoy	Lexer, Parser, Tree Walker and Documentation
Smridh Thapar	Testing, Back End and Documentation
Kishan Iyer	Lexer, Parser, Tree Walker and Documentation

Programming Style Guide

Introduction

Almost as important as writing quality, bug-free code is writing well commented and easy to read code. We knew that since we were all working in a team and would have to work on each other's code how important it would be to adhere to established coding formats. This includes comments and the location and use of curly braces, tabs, white space, and semicolons.

Antlr Style

Antlr has its own syntax and a different style of formatting from C and Java. The convention we followed was:

```
example :       foo
| bar
| foobar
;
```

Java Style Guide

The rest of the java code was formatted as follows:
Opening braces must be accompanied by closing braces at the same level of indentation at a lower. The following two fashions of brace and parenthesis placement accepted and are used by our team members.

```
Form1 {          Form 2
.                {.
.                  .
}                }
```

Comments

Comments are either single line or multi line comments. Each one of us made it a point to use comments wherever deemed necessary. Large explanations should be kept at the beginning of the file or must be clearly separated from other instructions.

Software Development Environment

It was left up to every one of us as to which IDE to use and whether to use Antlr as an external tool or as a plug-in. With the exception of Kangkook who worked on a Mac, the rest of us worked on Windows XP machines. All of us used the Eclipse IDE with the CVS plug-in.

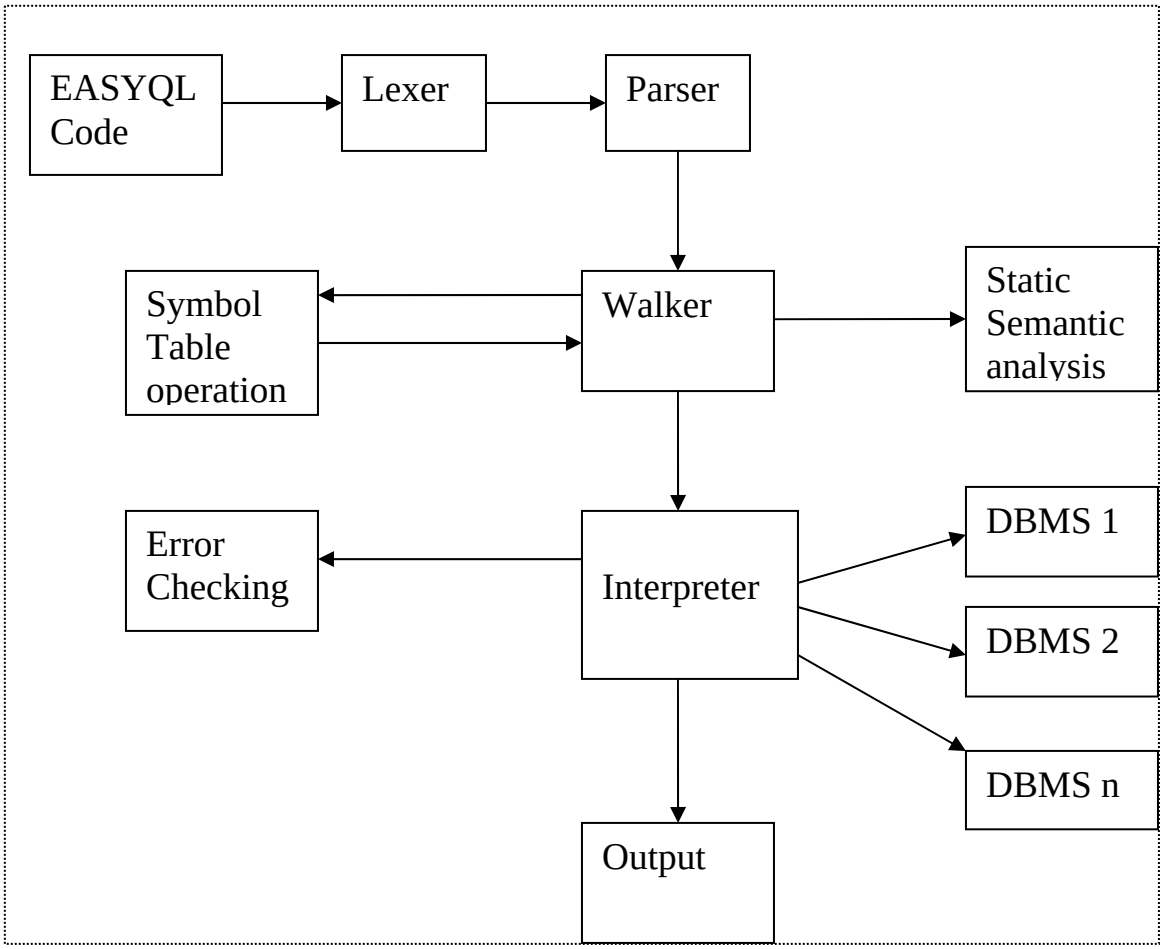
Project Timeline and Log

Group meeting to finalize project topic	15th Sept
Complete White Paper	25th Sept
White Paper submitted	26th Sept
Group meeting to discuss Language syntax and project scope	4th Oct
Group meeting to finalize syntax and divide work on the LRM	10th Oct
LRM submitted	19th Oct
CVS setup	21st Oct
Start work on the Lexer and Parser along with test programs	1st Nov
Crude lexer and Parser ready	14th Nov
Initial Interpreter ready	20th Nov
Modification of Parser to build AST	25th Nov
Meeting with TA, Deepti, to discuss progress made	26th Nov
Group meeting to discuss the background Java classes required and work on Tree Walker begun	30th Nov
Lexer and Parser complete. Few bugs yet to resolve	5th Dec
Crude Tree Walker ready with background java classes	8th Dec
Database Connectivity functionality added and static semantic analysis added	9th Dec
Revision to Interpreter and database connectivity code	11th Dec
Final version of Tree Walker ready. Documentation work begun	13th Dec
Final Code ready	17th Dec
Documentation and Presentation material ready	18th Dec

5. Architectural Design

Component Diagram

The EasyQL architectural design is illustrated below. The Interpreter is the central part of the language design. It connects the backend to the front end.



Interaction between Components

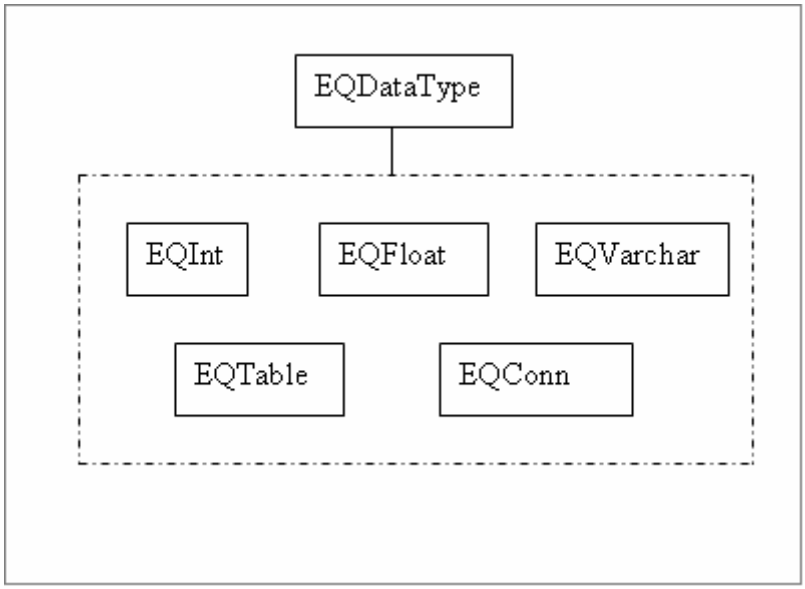
The Lexer is the component that reads the input file and creates tokens out of it. These tokens are given to the Parser which then creates an abstract syntax tree. The parser also checks for the syntax errors in the code and returns the errors to the user if the program has syntax errors. If the program syntax is correct the abstract syntax tree created by the parser is passed to the Tree Walker which recursively walks the AST and calls appropriate methods in the Interpreter class. For example, whenever an assignment is made the interpreter class is called which then checks the symbol table for the corresponding variable and performs the assignment. The interpreter performs static semantic analysis as well. In certain cases the tree walker directly calls other java classes which perform operations on the database instances. The symbol table has been implemented as a hash table.

The main() method is in the class Main. It first checks whether the input is coming from the console or a text file and correspondingly initializes the lexer, the parser and the tree walker in different ways. The class Main also handles exceptions and errors occurs in other classes, and prints corresponding messages.

Handling the different data types in EasyQL

The back end performs the task of handling the various data types supported by EasyQL. All the data are encapsulated in classes which are derived from the parent class EasyQLDataTypes. This class performs the basic operations unless they have been overridden by the derived class. Our simple data types such as int, varchar and float, are in classes EQInt, EQFloat and EQVarchar respectively. These classes are simply wrapper classes.

The specialized data types in our language, namely, Table and Connection, are implemented in classes EQTable and EQConn. These are also wrapper classes that contain the Java JDBC objects and perform the database connectivity.



Like temporary namespaces for regular variables, EasyQL has a temporary tablespace for the table variables. The use of this temporary tablespace is that the user can create and use temporary tables which are deleted at the end of the program.

6. Testing Plan

Incremental Testing

The EasyQL team understood that testing needed to be carried out throughout the development phase. Our plan during the project had been to test each component of our compiler once a change had been introduced. During the development of the parser for instance, we would perform tests each time a new rule was added to ensure that it was functioning as per our requirement.

Consequently, we did not need to perform any integration testing, as all the code was merely added to an existing common code base. Our test cases were input as files to our compiler. Before each team member updated our CVS repository with his changes, he would ensure that the changes implemented properly tested for correctness. Typically, one other team member would also test the changes made by some one.

The EasyQL Policy

Once major changes had been made to the compiler, we would perform a thorough test of each of our components to ensure it was working correctly. We would proceed with our work only when we were satisfied with the result of the tests.

Regression Testing

We frequently performed regressions tests to ensure that additions to our compiler did not, in any way, affect what was previously functioning correctly. This primarily involved running the previous test cases once again on the modified code.

Testing in Phases

Initially, while development of the parser, our tests involved checks to ensure that our rules had been well-defined and that statements in our language which are similar to that of Java were being accepted. This included variable declarations, if statements, comments and so on. Some of the testing files we used at this stage were:

```
int i,j;

i=0;
j=10;

while (j>i)
{
	display ("i is");
	display (i);
	display ("j is");
	display (j);

	i=i+1;

	if (i==10) {
		display ("i is 10, exiting");
	}
}
```

Once this was done, we had to ensure that statements which are unique to our language were also being accepted. We frequently ran into issues with non-determinism which were promptly fixed. For instance, there was a conflict between our “delete” and “desc” statements with a lookahead of 2. We had to make a call between increasing the lookahead to 3 or modifying our statement. The result – our “desc” statement has been modified to “dsc”.

When we had defined the structure of our AST, we ran tests on all parts of the language defined until that point, to check that the tree was constructed as we had desired. Each of these tests were repeated if a change had been made to our EasyQL.g file.

In the Tree Walker, to initially check that the tree was being walked properly, we would assign the tree nodes to local variables and output their values. Once we were satisfied with our back-end code, we would incorporate it within our walker and test our outputs:

Test Files

Test2.eql

```
connection c1;
int i1;
table t1,t2;


c1 = conn("69.22.220.234", "3306", "test", "mysql", "root","plt123");
i1=0;
t1.create (c1,"name varchar(255), age int");
t2=c1::t1;

while (i1<10) {
	i1=i1+1;
	display (i1);
	t2.insert["name,age"]("'Kishan',22");
}

display (t2);
```

test3.eql - another example for database connection

```
//declaration

int i1;
table T1,T2,T3;
connection C1,C2;

//initialization
i1=0;
C1 = conn("69.22.220.234", "3306", "test", "mysql", "root",
"plt123");
C2 = conn("localhost", "3306", "test", "mysql", "root", "plt123");
T1.create (C1,"name varchar(255), age int");
T2.create (C2,"name varchar(255), age int");

T1=C1::T1;
```

```
T2=C2::T2;

//program execution


while (i1<10) {
	i1=i1+1;
	display(i1);
	T1.insert["name,age"]("'Kishan',22");
	T2.insert["name,age"]("'Kishan',22");
}

display (T1);
display (T2);

display (T1["age"](""));
display (T2["name"](""));

T1.store (C1);
T2.store (C2);

display(C1::T1);
display(C2::T2);

create.eql

table T1;
int ret;

ret = T1.create (DefConn,"name varchar(255), age int");

if (ret)  {
	T1=DefConn::T1;

	display ("");
	display ("Tables in Default Connection");
	display (DefConn.list());

	display ("");
	display ("Description of T1");
	display (T1);
} else {
	display ("create table failed");
}
```

Finally, once we had the compiler working as planned, we continued to perform tests to make sure we had not missed anything previously. These tests uncovered some holes in our compiler. We realized that some basic arithmetic operations had not been properly defined, and that type checking was not being performed in certain areas.

No tools were used to perform these tests. We manually created test files and then fed them to our compiler to verified the results.

7. Lessons Learned

Kishan Iyer

I feel that our team worked really well once we began working on the project by sitting together and sharing our ideas. Our work may have progressed more smoothly had all four of us done this earlier, rather than the approach we adopted for most of the semester – divide responsibility, work on the individual components, and submit to the rest of the team for testing. Building the compiler would have been much quicker, had we been able to obtain feedback from the team instantly instead of waiting until the others could make time for it. We worked best when we worked as a team.

Saahil Peerbhoy

The most important lesson I learned from this project was working in a team. Since all of us stayed far apart, we had difficulty scheduling meetings at first, but soon enough we found certain days and time during the week which was compatible with everyone. Initially we worked on the model of dividing the work amongst us and delivering it at the deadline. However, we found that this system was not working well for us and we shifted to working together. I must also say that there were times when I thought that our project was going nowhere and that we would never finish it on time but I guess when you have a good team to back you up Our team had an excellent team leader in Kangkook and I learned a great deal from him as to how to manage a project.

Kangkook Jee

Coming back to college after working in a company for five years it took me some time getting used to a college environment again. Initially our work on the project progressed slowly but once we had the initial modules working it progressed much faster. Our team co-operated very well and all decisions were reached after a consensus. I only feel that we should have started work on our back end a little earlier as our project needed a lot of work on it and we might have been able to do a better job with it had we started early.

Smridh Thapar

Putting the computer theory into actual use has been an adventures learning experience of this project. Using the regular expressions to actually create an application which further lets users create programs has been a fulfilling experience for me being a computer science graduate. Creating a software of descent scale as this project over a short time span has also taught me project synchronization skills with integrations tools as CVS. Working on the backend of this project gave me a much clearer picture of JAVA jdbc connectivity and the interfacing of the ANTLR front end with a backend.

8. Project Code

Lexer.g

```
class EASYQLLexer extends Lexer;

options {
  exportVocab=EASYQL;
  charVocabulary = '\0'..'\377';
  testLiterals=false;    // don't automatically test for literals
  k=3;                   // two characters of lookahead
}

protected
LETTER
  : 'A' .. 'Z'
  | 'a' .. 'z'
  ;

protected
DIGIT
  : '0' .. '9'
  ;

NUMBER
  : (DIGIT)+
        ( '.' (DIGIT)* { $setType(FLOATLIT);}
        | /* empty */ { $setType(INTLIT); }
        )
;

ID options { testLiterals = true; }
: (LETTER | '_') (LETTER | DIGIT | '_')*
;

WS
  : ( ' '
    | '\t'
    | '\f'

    // handle newlines
    | ( "\r\n"  // DOS/Windows
      | '\r'    // Macintosh
      | '\n'    // Unix
      )
      // increment the line count in the scanner
      { newline(); }
    )
    { $setType(Token.SKIP); }
  ;

// single line comments
SL_COMMENT
  : "//" (~('\n'|'\r'))*
    { $setType(Token.SKIP); }
  ;
```

```
DOLLAR           : '$'   ;
DOT              : '.'   ;
COLON            : ':'   ;
SEMI             : ';'   ;
DCOLON           : "::"  ;
COMMA            : ','   ;
EQUALS           : '='   ;
LBRACKET         : '['   ;
RBRACKET         : ']'   ;
RCURLY           : '}'   ;
LCURLY           : '{'   ;
LPAREN           : '('   ;
RPAREN           : ')'   ;
NOT_EQUALS       : "!="  ;
LT               : '<'   ;
LTE              : "<="  ;
GT               : '>'   ;
GTE              : ">="  ;
PLUS             : '+'   ;
MINUS            : '-'   ;
MULT             : '*'   ;
DIV              : '/'   ;
MOD              : '%'   ;
PLUSPLUS         : "++"  ;
MINUSMINUS       : "--"  ;
// logical operators
AND              : "&&" ;
OR               : "||" ;
NOT              : "!" ;
LEQUALS          : "==" ;

UPDATE           : ".update";
INSERT           : ".insert";
DELETE           : ".delete";
DROP             : ".drop";
CREATE           : ".create";
LIST             : ".list";
NEXT         : ".next";
RESET        : ".reset";
DESC             : ".dsc";
DISP             : ".display";
STORE            : ".store";


  // multiple-line comments
ML_COMMENT
  : "/*"
    (               /* '\r' '\n' can be matched in one alternative
or by matching
                       '\r' in one iteration and '\n' in another. I
am trying to
                       handle any flavor of newline that comes in,
but the language
                       that allows both "\r\n" and "\r" and "\n" to
all be valid
                       newline is ambiguous. Consequently, the
resulting grammar
                       must be ambiguous. I'm shutting this warning
off.
```

```
                    */
      options {
        generateAmbigWarnings=false;
      }
      :  { LA(2)!='/' }? '*'
      | '\r' '\n' {newline();}
      | '\r' {newline();}
      | '\n' {newline();}
      | ~('*'|'\n'|'\r')
    )*
    "*/"
    {$setType(Token.SKIP);}
;

CHARLIT
  : '\''! . '\''!
  ;

STRING_LITERAL
  : '"'!
    ( '"' '"'!
    | ~('"'|'\n'|'\r')
    )*
    ( '"'!
    | // nothing -- write error message
    )
   ;

/*
grammar.g
$Id: EasyQL.g,v 1.50 2006/12/18 20:05:21 ki2147 Exp $
*/
/*         header {
            // use this area to specify a package for the resulting
parser/lexer
             // and any imported classes
         package EasyQL.antlr;
         }
 */
         class EASYQLParser extends Parser;
         options {
                 exportVocab=EASYQL;
                 k=2;
                 buildAST=true;
         }

         tokens {
                 PROGRAM;
                 BLOCK;
                 PAREN;
                 TABLE_EXPR;
                 TABLE_ATTR;
                 TABLE_COND;
                 TABLE_UPDATE;
                 TABLE_INSERT;
                 TABLE_DELETE;
                 TABLE_DROP;
                 TABLE_CREATE;
                 TABLE_STORE;
```

```
                TABLE_NEXT;
                TABLE_RESET;
            TABLE_FETCH;
                DSC;
                List;
                CONN_EXPR;
                FUNC_CALL;
                CONDLIST;
                ASSIGNLIST;
                UMINUS;
        }

        program
                :
                        (statement)* EOF!
                         {#program = #([PROGRAM,"PROGRAM"],#program);}
                ;

        // FIXME: now it is set to have declaration of a variable
        // any place in the language like java or shell scripts
        // We can think more about this

         statement
                 : assignment
                 | while_statement
                 | if_statement
                 | block
                 | decl
                 | expr SEMI!
                 ;

        cl_statement
                : statement
                | "exit"         {System.exit(0);}
                | EOF!                {System.exit(0);}
                ;

         //FIXME: now it is left associative
         //need to be corrected as right associative

         assignment
                 : (ID EQUALS^)+ expr
                   SEMI!
                 ;

  /*
         table_assignment
                 : table_expr EQUALS^
                   (table_expr|conn_expr)
                   SEMI!
                 ;
  */

          while_statement
                 :        "while"^ LPAREN! arith_expr RPAREN! statement
                 ;

        if_statement
             :   "if"^ LPAREN! arith_expr RPAREN! statement
```

```
                (
                        options { greedy= true; }:
                        "else"! statement
                )?
    ;

 decl
        : (         "int"^
                  |"float"^
                  |"varchar"^
                  |"date"^
                  |"table"^
                  |"connection"^
          ) identlist SEMI!
        ;

expr
        : arith_expr
        | table_expr
        | conn_expr
        | func_call

        //{#expr = #([EXPR,"EXPR"],#expr);}
        ;

arith_expr
        : expr_not ( ( AND^ | OR^) expr_not)?
        ;

expr_not
        : (NOT^)? expr_compare
        ;

expr_compare
        : expr_add_sub (( LEQUALS ^
        | NOT_EQUALS^
        | GT^
        | LT^
        | GTE^
        | LTE^
        ) expr_add_sub)?
        ;

expr_add_sub
        : expr_mul_div ( (PLUS^ expr_mul_div)
        | (MINUS^ expr_mul_div)
        )*
        ;

expr_mul_div
        : expr_unary ( (MULT^ expr_unary)
        | (MOD^ expr_unary)
        | (DIV^ expr_unary)
        )*
        ;

expr_unary
        : (MINUS! (INTLIT|FLOATLIT|ID))
        { #expr_unary = #([UMINUS,"UMINUS"],#expr_unary);}
```

```
                | atom

                ;

        //FIXME: not sure it is ok to have CHARLIT and STRING_LITERALs in
atom
        atom
                : LPAREN! expr RPAREN!
                //{#atom = #([BLOCK,"PAREN"],#atom);}
                | NUMBER
                | INTLIT
                | FLOATLIT
                | CHARLIT
                | STRING_LITERAL
                | ID
                ;

        //FIXME: need to think more about tree walker structure
        // need to think more about conditional statements here (too
generic)

        table_expr
                : ID^ table_attr (table_cond)
                  {#table_expr =
#([TABLE_EXPR,"TABLE_EXPR"],#table_expr);}
                | table_update
                | table_insert
                | table_delete
                | table_drop
                | table_create
                | table_store
                | table_next
                | table_reset
      //    | table_fetch
                | desc
                ;

        //FIXME: need to think about having nothing as an table_attr
        //STRING_LITERAL was (identlist)..also removed the '|' part
        table_attr
                : LBRACKET! STRING_LITERAL
                {#table_attr = #([TABLE_ATTR,"TABLE_ATTR"],#table_attr);}
                 RBRACKET!
                // | /* nothing */
                ;

        //STRING_LITERAL was condlist
        table_cond
                : LPAREN! STRING_LITERAL RPAREN!
                {#table_cond = #([TABLE_COND,"TABLE_COND"],#table_cond);}
                | /* nothing */
                ;

        conn_expr
                ://ID DCOLON^ (ID|table_expr)
                 ID DCOLON! ID
                 {#conn_expr = #([CONN_EXPR,"CONN_EXPR"],#conn_expr);}
                |list
                ;
```

```
        table_update
                 : ID UPDATE! table_attr table_cond
                         {#table_update =
#([TABLE_UPDATE,"TABLE_UPDATE"],#table_update);}
                 ;

        table_insert
                 :ID INSERT! table_attr table_cond
                 {#table_insert =
#([TABLE_INSERT,"TABLE_INSERT"],#table_insert);}
                 ;

         //table_delete
             //    :        ID DELETE! LPAREN! condlist RPAREN!
             //    {#table_delete =
#([TABLE_DELETE,"TABLE_DELETE"],#table_delete);}
             //    ;

        table_delete
                  :     ID DELETE! table_cond
                     {#table_delete =
#([TABLE_DELETE,"TABLE_DELETE"],#table_delete);}
                ;


        table_drop
                 :        ID DROP! LPAREN! RPAREN!
                 {#table_drop =
#([TABLE_DROP,"TABLE_DROP"],#table_drop);}
                 ;

        table_create
            : ID CREATE! LPAREN! ID COMMA! STRING_LITERAL RPAREN!
                  {#table_create =
#([TABLE_CREATE,"TABLE_CREATE"],#table_create);}
            ;

        table_store
                 :        ID STORE! LPAREN! ID RPAREN!
                 {#table_store =
#([TABLE_STORE,"TABLE_STORE"],#table_store);}
                 ;

            table_next
            :     ID NEXT! LPAREN! RPAREN!
                  {#table_next = #([TABLE_NEXT,
"TABLE_NEXT"],#table_next);}
            ;

            table_reset
            :     ID RESET! LPAREN! RPAREN!
                  {#table_reset = #([TABLE_RESET,
"TABLE_RESET"],#table_reset);}
            ;



         desc
```

```
        :         ID DESC! LPAREN!  RPAREN!
                 {#desc = #([DSC,"DSC"],#desc);}
                 ;


      //table_fetch
      //    : "fetch" LPAREN! ID table_attr RPAREN!
      //          {#table_fetch = #([TABLE_FETCH,"TABLE_FETCH"],
#table_fetch);}
      //    ;

         //FIXME: need to think about display table_expr, expression ...

      func_call
                 :        ((("conn"|"display") LPAREN! arglist
RPAREN!)|("fetch" LPAREN! ID table_attr RPAREN!))
                 {#func_call = #([FUNC_CALL,"FUNC_CALL"],#func_call);}
                ;

         list
                 :        ID LIST! LPAREN!  RPAREN!
                 {#list = #([List,"List"],#list);}
                 ;

        arg
                : expr
                ;

        arglist
                : arg (COMMA! arg)*
                | /* nothing */
                ;

        identlist
                :         ID (COMMA! ID)*
                ;

        condlist
                :          cond (COMMA! cond)*
                          {#condlist = #([CONDLIST,"CONDLIST"],#condlist);}
                ;
        cond
                :        ID (GT^ |GTE^ |LEQUALS^ |NOT_EQUALS^ |LT^ |LTE^
)(arith_expr|table_expr)
                ;

        assignlist
                :         assign (COMMA! assign )*
                         {#assignlist =
#([ASSIGNLIST,"ASSIGNLIST"],#assignlist);}
                ;

         assign
                :        ID EQUALS^ expr
                ;

          block
            :   LCURLY! (statement)* RCURLY!
            {#block = #([BLOCK,"BLOCK"],#block);}
```

```
            ;
```

Tree Walker.g

```
class EASYQLTreeWalker extends TreeParser;

options {
      importVocab=EASYQL;
}

{
      EQInterpreter itpr = new EQInterpreter();
      static EQDataType null_data = new EQDataType( "<NULL>" );
}

expr returns [EQDataType r]
{     EQDataType a,b;
      r = null_data;
}
      :
      #(PROGRAM (stmt:. {
                                    r = expr(#stmt);
                                    }
                        )*)
      //declaration
      |   #("int" (intid:ID         {
                                                a = new EQInt();

      a.setName(intid.getText());
                                                r =
itpr.setVariable(a);
                                          }
                        )*
                  )

      |     #("float" (floatid:ID   {
                                                a = new EQFloat();

      a.setName(floatid.getText());
                                                r =
itpr.setVariable(a);

                                          }
                              )*
                  )

      |     #("varchar" (varcharid: ID{
                                                a = new
EQVarchar();

      a.setName(varcharid.getText());
                                                r =
itpr.setVariable(a);
                                                }
                              )*
            )
```

```
      |     #("connection" (connid: ID {
                                                a = new EQConn();

      a.setName(connid.getText());
                                                r =
itpr.setVariable(a);
                                                }
                              )*
            )
      |     #("table" (tblid: ID {
                                                a = new EQTable();

      a.setName(tblid.getText());
                                                r =
itpr.setVariable(a);
                                                }
                              )*
            )

      //unary minus..added by saahil
      |     #(UMINUS a=expr)              {r=a.uminus();}

      //arithmetic operations
    |   #(PLUS  a=expr b=expr)  {r = a.plus(b);}

    |   #(MINUS a=expr b=expr)  {r = a.minus(b);}
    |   #(MULT  a=expr b=expr)  {r = a.mult(b);}
    | #(DIV a=expr b=expr)    {r = a.div(b); }
    //logical operations
    | #(GT a=expr b=expr)     {r = a.gt(b); }
    | #(GTE a=expr b=expr)    {r = a.gte(b); }
    | #(LT a=expr b=expr)     {r = a.lt(b); }
    | #(LTE a=expr b=expr)    { r = a.lte(b); }
    | #(LEQUALS a=expr b=expr){ r = a.lequals(b); }
    | #(NOT a=expr)                  {r = a.not(); }
    | #(NOT_EQUALS a=expr b=expr)   {r= a.ntequals(b);}
    |   #(AND a=expr b=expr)    {r = a.and(b);}
    |   #(OR a=expr b=expr)   {r = a.or(b);}

    // assignment
    | #(EQUALS a=expr b=expr) {
                                                r = itpr.assignValue
(a,b);
                                                try {

      java.lang.Thread.sleep (50);
                                                } catch (Exception
tempe){
                                                }
                                          }

    //Table Expressions..saahil
    |#(TABLE_EXPR #(tablid:ID #(TABLE_ATTR attlist: STRING_LITERAL)
(#(TABLE_COND condlist: STRING_LITERAL))?))
      {
                  a=itpr.getVariable(tablid.getText());
                  if(a instanceof EQTable)
                  {
```

```
      r=((EQTable)a).query(attlist.getText(),condlist.getText());
                  }
                  else {
                        System.err.println("[EasyQL][TreeWalker]error in
the delete");
                        System.exit (0);
                  }
      }

    |#(TABLE_DELETE tablid1:ID #(TABLE_COND condlist1:STRING_LITERAL))
      {
            a=itpr.getVariable(tablid1.getText());
                  if(a instanceof EQTable)
                  {
                        r=((EQTable)a).delete(condlist1.getText());
                  }
                  else {
                        System.err.println("[EasyQL][TreeWalker]error in
the delete");
                        System.exit (0);
                  }
      }

    |#(TABLE_INSERT tablid2:ID #(TABLE_ATTR ins_attlist:STRING_LITERAL)
#(TABLE_COND ins_condlist: STRING_LITERAL))
      {
            a=itpr.getVariable(tablid2.getText());
            if(a instanceof EQTable)
            {

      r=((EQTable)a).insert(ins_attlist.getText(),ins_condlist.getText())
;
            }
            else {
                        System.err.println("[EasyQL][TreeWalker]error in
the delete");
                        System.exit (0);
                  }
      }

    |#(TABLE_UPDATE tablid3:ID #(TABLE_ATTR upd_attlist:STRING_LITERAL)
#(TABLE_COND upd_condlist: STRING_LITERAL))
      {
            a=itpr.getVariable(tablid3.getText());
            if(a instanceof EQTable)
            {

      r=((EQTable)a).update(upd_attlist.getText(),upd_condlist.getText())
;
            }
            else {
                        System.err.println("[EasyQL][TreeWalker]error in
the update");
                        System.exit (0);
                  }
      }

      |#(TABLE_DROP drop_id:ID)
```

```
      {
            a=itpr.getVariable(drop_id.getText());
            if(a instanceof EQTable)
            {
                  r=((EQTable)drop_id).drop();
            }

            else {
                        System.err.println("[EasyQL][TreeWalker]error in
the drop");
                        System.exit (0);
                  }
      }


    |#(DSC desc_id:ID)
      {
            a=itpr.getVariable(desc_id.getText());
            if(a instanceof EQTable)
            {
                  r=((EQTable)a).describe();
            }
            else {
                        System.err.println("[EasyQL][TreeWalker]error in
the desc");
                        System.exit (0);
                  }
      }


      |#(TABLE_CREATE tab_id:ID conn_id:ID create_string:STRING_LITERAL)
      {
            a=itpr.getVariable(tab_id.getText());
            b=itpr.getVariable(conn_id.getText());
            if((a instanceof EQTable) && (b instanceof EQConn))
            {
                  //System.out.println("LHS is Table type and Connecion's
fine");

      r=((EQTable)a).create((EQConn)b,tab_id.getText(),create_string.getT
ext());
            }
            else {
                        System.err.println("[EasyQL][TreeWalker]error in
the create");
                        System.exit (0);
                  }
      }

      |#(TABLE_STORE tabl_id: ID con_id: ID)
      {
            a=itpr.getVariable(tabl_id.getText());
            b=itpr.getVariable(con_id.getText());
            if((a instanceof EQTable) && (b instanceof EQConn))
            {
                  r=((EQTable)a).store((EQConn)b);
            }
            else {
```

```
                        System.err.println("[EasyQL][TreeWalker]error in
the store");
                        System.exit (0);
                  }
      }


      |#(TABLE_NEXT next_id: ID)
      {
            a=itpr.getVariable(next_id.getText());
            if(a instanceof EQTable)
            {
                  r=((EQTable)a).next();  //hey smridh uncomment this
when u make the function
            }
            else {
                        System.err.println("[EasyQL][TreeWalker]error in
the next");
                        System.exit (0);
                  }
      }

      |#(TABLE_RESET reset_id: ID)
      {
            a=itpr.getVariable(reset_id.getText());
            if(a instanceof EQTable)
            {
                  r=((EQTable)a).reset();  //hey smridh uncomment this
when u make the function
            }
            else
            System.out.println("error in the reset function");
      }


    |#(CONN_EXPR a=expr tablename:ID)
      {
            if(a instanceof EQConn)
            {

                  r=((EQConn)a).getTable(tablename.getText());

            }
            else a.error("error in CONN_EXPR");
      }



    // ID
    | id:ID                         { r = itpr.getVariable(
id.getText()); }
    //number
    | intlit:INTLIT                 {r=itpr.getInt(intlit.getText());}
    | floatlit:FLOATLIT
      {r=itpr.getFloat(floatlit.getText());}

    //varchar
    | varchar:STRING_LITERAL  {r=itpr.getVarchar(varchar.getText());}
```

```
     //if statement control
     |  #("if" a=expr thenp:. (elsep:.)?)
             {
                 if ( !( a instanceof EQInt ) )
                     return a.error( "if: expression should be bool" );
                 if ( ((EQInt)a).var!=0 )
                     r = expr( #thenp );
                 else if ( elsep != null)
                     r = expr( #elsep );
                   }

       //while statement ..saahil
       |#("while" eval:. whilebody:.)
                   {
                a= expr(#eval);

                        if ( !( a instanceof EQInt ) )
                     return a.error( "while: expression should be bool" );

               int i = ((EQInt)a).var;

                 while ( i!=0 ) {
                     r = expr( #whilebody );
                     a= expr(#eval);
                     if ( !( a instanceof EQInt ) )
                         return a.error( "while: expression should be
bool" );
                     i = ((EQInt)a).var;
                 }
                   }


       | #(BLOCK (statement:.  {
                               r=expr(#statement);
                                      }
                   )*
       )

      | #(FUNC_CALL
             (func_id:. args:.)
             {
                   if (func_id.getText().startsWith("conn")) {
                         AST temp = null;

                         String addr = #args.getText();
                         #args = #args.getNextSibling();
                         String port = #args.getText();
                         #args = #args.getNextSibling();
                         String dbname = #args.getText();
                         #args = #args.getNextSibling();
                         String dbtype = #args.getText();
                         #args = #args.getNextSibling();
                         String uid = #args.getText();
                         #args = #args.getNextSibling();
                         String passwd = #args.getText();

                         //System.out.println
(addr+":"+port+":"+dbname+":"+dbtype+":"+uid+":"+passwd);
```

```
                        r = itpr.getConnection
(addr,port,dbname,dbtype,uid,passwd);

                } else if (func_id.getText().startsWith("display")) {

                        a = expr (#args);
                        r = itpr.display(a);
                }
                else if (func_id.getText().startsWith("fetch")){

                        String table_id = #args.getText();
                        #args = #args.getNextSibling();
                        #args = #args.getFirstChild();
                        String att_name = #args.getText();
                        a=itpr.getVariable(table_id);
                        r=((EQTable)a).fetch(att_name);
                }
          }
    )
   |#(List a=expr) {
    if (a instanceof EQConn) {
          r = ((EQConn) a).listTable();
    } else
          a.error (a,"LIST");
   }

   |#(Display a=expr) {
    if (a instanceof EQTable) {
          r = ((EQTable) a).display();
    } else
          a.error (a,"Unknown Table");
   }
   ;
```

EQConn.java

```
import java.io.PrintWriter;
import java.sql.*;

public class EQConn extends EQDataType {
public Connection var;
public static final String query_all = "select * from ";

public EQConn(String a, Connection b) {
      this.name = a;
      var = b;
}

public EQConn (Connection a) {
      this.var = a;
}

public EQConn () {
      this.var = null;
}

public String typename() {
```

```
        return "Conn";
    }

public EQConn copy (){
      return new EQConn (var);
}

public EQTable getTable (String tablename) {

      EQTable ret = null;

      try {
            Statement stmt = var.createStatement();

            ResultSet rs = stmt.executeQuery (query_all+tablename);
            ret = new EQTable(rs);

      } catch (SQLException e1) {
            System.err.println("[EasyQL exception][EQConn.getTable]:
table name doesn't exist");
            //e1.printStackTrace();
      }

      return ret;
    }

public EQTable listTable() {
      EQTable tab = null;
      DatabaseMetaData metadata = null;
      try {
            metadata = var.getMetaData();
            String[] names = {"TABLE"};
            ResultSet tableNames = metadata.getTables(null,"%", "%",
names);

            tab = new EQTable(tableNames);
      }
      catch (SQLException e) {
            System.err.println("[EasyQL exception][EQConn.listTable]:
metadata retreival failed");
            //System.out.println("Exception in getDatabaseMetadata() " +
e);
      }

      return tab;
      }

      public void print(PrintWriter w) {
          if ( name != null )
              w.print( name + " = " );
          w.println(var);
      }
}
```

EQDataType.java

```
import java.io.PrintWriter;
```

```
public class EQDataType {
      String name;

    public EQDataType() {
        name = null;
    }

    public EQDataType ( String name ) {
        this.name = name;
    }

    public void setName( String name ) {
        this.name = name;
    }

    public String getName() {
      return this.name;
    }

    public String typename() {
        return "unknown";
    }

    public EQDataType error( String msg ) {
        throw new EQException( "illegal operation: " + msg
                               + "( <" + typename() + "> "
                               + ( name != null ? name : "<?>" )
                               + " )" );
    }

    public EQDataType error( EQDataType b, String msg ) {
        if ( null == b )
            return error( msg );
        throw new EQException(
            "illegal operation: " + msg
            + "( <" + typename() + "> "
            + ( name != null ? name : "<?>" )
            + " and "
            + "<" + typename() + "> "
            + ( name != null ? name : "<?>" )
            + " )" );
    }

    public EQDataType plus( EQDataType b ) {
        return error( b, "+" );
    }

    //unary opern..added by saahil
    public EQDataType uminus(  ) {
        return error( this.name );
    }

    public EQDataType minus( EQDataType b ) {
        return error( b, "-" );
    }

    public EQDataType mult( EQDataType b ) {
        return error( b, "*" );
```

```
    }

    public EQDataType div( EQDataType b ) {
        return error( b, "/" );
    }

    public EQDataType mod( EQDataType b ) {
        return error( b, "%" );
    }

    public EQInt gt( EQDataType b ) {
        return (EQInt) error( b, ">=" );
    }
    public EQInt gte( EQDataType b ) {
        return (EQInt) error( b, ">=" );
    }
    public EQInt lt( EQDataType b ) {
        return (EQInt) error( b, "<" );
    }
    public EQInt lte( EQDataType b ) {
        return (EQInt) error( b, "<=" );
    }
    public EQInt lequals( EQDataType b ) {
        return (EQInt) error( b, "==" );
    }
    public EQInt and ( EQDataType b) {
      return (EQInt) error (b, "&&");
    }
    public EQInt or ( EQDataType b) {
      return (EQInt) error (b, "||");
    }

    //saahil..adding '!='
    public EQInt ntequals( EQDataType b ) {
        return (EQInt) error( b, "!=" );
    }

    public EQInt not () {
        return (EQInt) error("!");
    }
    public EQDataType copy() {
        return new EQDataType();
    }
    public void what (PrintWriter w) {
        w.print( "<" + typename() + ">  " );
        print ( w );
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " ?= " );
        w.println( "<undefined>" );
    }

    public void print() {
        print( new PrintWriter( System.out, true ) );
    }

    public EQInt display () {
```

```
        EQInt ret = new EQInt (0);
        System.out.println ("EQDataType: display()");
        return ret;
    }
}
```

EQFloat.java

```
import java.io.PrintWriter;

public class EQFloat extends EQDataType {
      double var;

      public EQFloat () {
            super();
      }

      public EQFloat (double a) {
            this.var=a;
      }

      public String typename() {
        return "Float";
    }

    public static double floatValue( EQDataType b) {
        if ( b instanceof EQFloat )
            return ((EQFloat) b).var;
        if ( b instanceof EQInt )
            return (double) ((EQInt) b).var;
        return 0;

    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.println( Double.toString( var ) );
    }

    public EQFloat uminus ()
    {
      return new EQFloat(-var);
    }

    public EQDataType plus ( EQDataType  b ) {
        return new EQFloat( var + EQFloat.floatValue(b) );
    }

    public EQDataType minus ( EQDataType  b ) {
        return new EQFloat( var - EQFloat.floatValue(b) );
    }

    public EQDataType mult ( EQDataType  b ) {
        return new EQFloat( var * EQFloat.floatValue(b) );
    }
```

```
    public EQDataType div ( EQDataType  b ) {
        return new EQFloat( var / EQFloat.floatValue(b) );
    }

    public EQDataType mod ( EQDataType  b ) {
        return new EQFloat( var % EQFloat.floatValue(b) );
    }

    public EQInt gt( EQDataType b ) {
        return new EQInt( (var > EQFloat.floatValue(b))?1:0);
    }

    //saahil..adding "!="
    public EQInt ntequals( EQDataType b ) {
        return new EQInt( (var == EQFloat.floatValue(b))?0:1);
    }

    public EQInt gte( EQDataType b ) {
        return new EQInt( (var >= EQFloat.floatValue(b))?1:0);
    }
    public EQInt lt( EQDataType b ) {
        return new EQInt( (var < EQFloat.floatValue(b))?1:0);
    }
    public EQInt lte( EQDataType b ) {
        return new EQInt( (var <= EQFloat.floatValue(b))?1:0);
    }
    public EQInt lequals( EQDataType b ) {
        return new EQInt( (var == EQFloat.floatValue(b))?1:0);
    }
    public EQInt not () {
        return new EQInt( (var!=0)?0:1);
    }
    public EQFloat copy (){
      return new EQFloat (var);
    }
}
```

EQInt.java

```
import java.io.PrintWriter;

public class EQInt extends EQDataType {
      public int var;

      public EQInt () {
            super ();
      }

      public EQInt (int a) {
            this.var =a;
      }

      public String typename() {
        return "Int";
    }
```

```
public static int intValue( EQDataType b ) {
    if ( b instanceof EQInt )
        return ((EQInt) b).var;
    if ( b instanceof EQFloat )
        return (int) ((EQFloat) b).var;
    return 0;
}

public void print( PrintWriter w ) {
    if ( name != null )
        w.print( name + " = " );
    w.println( Integer.toString( var ) );
}

public EQDataType plus ( EQDataType  b ) {
    //System.out.print("\nEQInt function being used\n");
  if ( b instanceof EQInt )
        {
              return new EQInt( var + intValue(b) );
        }
    else if(b instanceof EQFloat)
        {
              return new EQInt((int)( var + EQFloat.floatValue(b) ));
        }
        else
        {
              return this.error("incompatible types");
        }
}
/*added by saahil..unary operation
public EQDataType minus() {

              return new EQInt( -var);
}
*/
public EQInt uminus ()
{
  return new EQInt(-var);
}

public EQDataType minus( EQDataType  b ) {
    if ( b instanceof EQInt )
        {
              return new EQInt( var - intValue(b) );
        }
    else if (b instanceof EQFloat)
        {
              return new EQInt((int)( var - EQFloat.floatValue(b) ));
        }
    else{
        return this.error("incompatible types");
        }
}

public EQDataType mult ( EQDataType  b ) {
  if ( b instanceof EQInt )
    {
        return new EQInt( var * intValue(b) );
```

```
        }
    else if (b instanceof EQFloat)
      {
            return new EQInt((int)( var * EQFloat.floatValue(b) ));
      }
    else{
      return this.error("incompatible types");
      }
    }

    public EQDataType div( EQDataType  b ) {
      if ( b instanceof EQInt )
        {
            return new EQInt( var / intValue(b) );
        }
    else if (b instanceof EQFloat)
      {
            return new EQInt((int)( var / EQFloat.floatValue(b) ));
      }
    else{
      return this.error("incompatible types");
      }
    }

    public EQDataType mod( EQDataType  b ) {
      if ( b instanceof EQInt )
        {
            return new EQInt( var % intValue(b) );
        }
    else if (b instanceof EQFloat)
      {
            return new EQInt((int)( var % EQFloat.floatValue(b) ));
      }
    else{
      return this.error("incompatible types");
    }
}

    public EQInt gt( EQDataType b ) {
      if((b instanceof EQInt)||(b instanceof EQFloat))
        {
            return new EQInt( (var > EQInt.intValue(b))?1:0);
        }
      else{
            return (EQInt)this.error("incompatible types");
      }
    }

    //saahil..adding "!="
    public EQInt ntequals( EQDataType b ) {
        return new EQInt( (var == EQInt.intValue(b))?0:1);
    }


    public EQInt gte( EQDataType b ) {
        return new EQInt( (var >= EQInt.intValue(b))?1:0);
    }
    public EQInt lt( EQDataType b ) {
        return new EQInt( (var < EQInt.intValue(b))?1:0);
```

```
    }
    public EQInt lte( EQDataType b ) {
        return new EQInt( (var <= EQInt.intValue(b))?1:0);
    }
    public EQInt lequals( EQDataType b ) {
        return new EQInt( (var == EQInt.intValue(b))?1:0);
    }
    public EQInt not () {
        return new EQInt( (var!=0)?0:1);
    }

    public EQInt and( EQDataType b) {
      boolean x;
      x= ((var>0)&&((EQInt.intValue(b))>0))?true:false;
      return new EQInt( (x==true)?1:0);
    }

    public EQInt or( EQDataType b) {
      boolean x;
      x= ((var>0)||((EQInt.intValue(b))>0))?true:false;
      return new EQInt( (x==true)?1:0);
    }


    public EQInt copy (){
      return new EQInt (var);
    }
}
```

EQInterpreter.java

```
import java.sql.*;

public class EQInterpreter {
      private static final String db_name="EQL_DB";
      private static final String db_conninfo =
"jdbc:mysql://localhost:3306/";
      private static final String db_id="root";
      private static final String db_passwd="plt123";

      //private static final String db_conninfo =
"jdbc:mysql://69.22.220.234:3306/";

      EQSymTable symt;

      public EQInterpreter() {
            super();
            this.symt = new EQSymTable();
            initDB();
      }

    public boolean canProceed() {
      return true;
        //return control == fc_none;
    }
```

```
	public EQDataType assignValue (EQDataType a, EQDataType b) {
		if((a.typename() == b.typename()) ||
((a.typename()=="Float")&&(b.typename()=="Int")) ||
(a.typename().equals("Table")))
		{
			if((a.typename()=="Float")&&(b.typename()=="Int")) {
				if(null != a.name) {
					EQInt y = (EQInt)b;
					EQFloat x = new EQFloat(y.var);
					x.setName(a.name);
					symt.setValue(x.name,x);
					return x;
				}
				return a.error( b, "=" );
			}

			if (a.typename()=="Table" && b.typename()=="Table") {
				EQTable x;
				clearTable ((EQTable)a);

				x = createTable (((EQTable)a).name,(EQTable)b);
				x = ((EQTable)b).insertTable (x.var);

				symt.setValue(x.name,x);
			}

			if ( null != a.name )
		{
			EQDataType x;

				if (b.name==null) //means it is just a number
					x = b;
				else
			{
				x = b.copy();	//means it is an ID
			}
				x.setName ( a.name );
				symt.setValue (x.name, x);

			return x;
		}
		return a.error( b, "=" );
	}

	return a.error("Types are incompatible");
}

	//used only the first time while declaring a variable and checks
whether a variable
	//with the same name has been previously defined
	public EQDataType setVariable( EQDataType a) {
		if(symt.getValue(a.name)!=null)
		{
			return a.error("Variable "+a.name+" has been multiply
defined");
		}
		else{
			symt.setValue(a.name, a);
			return a;
```

```
		}
	}

	/*used whenever an assignment is made to a previously declared
object
	public EQDataType setVariable ( EQDataType a) {
		symt.setValue(a.name, a);
		return a;
	}*/

	public EQDataType getVariable (String s) {
		EQDataType x;
		x = symt.getValue(s);
		if (x==null) {
			x = new EQVariable(s);
			return x.error ("ID" +s+": not previously defined");
		}
		return x;
	}

	public EQInt getInt(String s) {
		return new EQInt (Integer.parseInt(s));
	}

	public EQFloat getFloat(String s) {
		return new EQFloat (Double.parseDouble(s));
	}

	public EQVarchar getVarchar(String s){
		return new EQVarchar ((String)s);
	}

	public EQConn getConnection (String host, String port, String
dbname, String dbtype, String id,String passwd) {
		EQConn ret;

		if (dbtype.toLowerCase().equals("mysql")) {
			ret = new EQMySQLConn (host,port,dbname,id,passwd);
		}
		//We can add more DBMS connections here
		else {
			ret =new EQConn ();
			ret.error("incorrect dbname specified");
		}
		return ret;
	}

	public EQInt display (EQDataType a) {
		EQInt ret = new EQInt (1);
		try {
			a.print ();
		} catch (Exception e) {
			ret.var = 0;
		}
		return ret;
	}

	private void initDB() {
```

```
            Connection con = null;
            EQConn defaultConn = null;

            try {
                  Class.forName("com.mysql.jdbc.Driver");
                  con =
DriverManager.getConnection(EQInterpreter.db_conninfo+"mysql",
EQInterpreter.db_id, EQInterpreter.db_passwd);
                  Statement stmt = con.createStatement();

                  String query = "show databases";
                  ResultSet rs = stmt.executeQuery(query);

                  boolean hasName = false;

                  while (rs.next()){
                        String dbname = rs.getString(1);
                        if (dbname.equals(EQInterpreter.db_name)) {
                              hasName =true;
                              break;
                        }
                  }

                  if (!hasName) {
                        query = "create database "+EQInterpreter.db_name;
                        stmt.execute (query);
                  } else {
                        query = "drop database "+EQInterpreter.db_name;
                        stmt.execute (query);
                        query = "create database "+EQInterpreter.db_name;
                        stmt.execute (query);
                  }

                  con.close();

            } catch (Exception e1) {
                  System.err.println("[EasyQL
error][EQInterpreter.initDB]: init DB failure");
                  System.exit (0);
            }

            try {
                  con =
DriverManager.getConnection(EQInterpreter.db_conninfo+EQInterpreter.db_na
me, EQInterpreter.db_id, EQInterpreter.db_passwd);
            } catch (Exception e2) {
                  System.err.println("[EasyQL
error][EQInterpreter.display]: default connection initialization
failure");
                  //e2.printStackTrace();
            }
            //generate a new connection info and put it into symbol table
            defaultConn = new EQConn ("DefConn",con);
            this.symt.setValue ("DefConn",defaultConn);
      }

      private EQTable createTable (String tname, EQTable a) {
            EQConn defConn =(EQConn) symt.getValue("DefConn");
            return createTable (defConn,tname, a);
```

```
	}

	private EQTable createTable (EQConn a, String tname, EQTable b) {

		EQTable ret=null;

		if (b.var==null || b.md==null)
			b.error("creatTable error - ResultSet, Metadata not
initialized");

		EQTableMD md = b.md;
		int columnSize = md.size();
		EQColumn column = null;

		String select_query  = "select * from "+tname;
		String create_query  = "create table "+tname+ " (";
		int i;

		for (i=0;i<columnSize-1;i++) {

			column = md.get(i);
			create_query+=column.columnName;
			switch (column.columnType) {
				case 1://case of vachar
				case 12:
					create_query+=" varchar(255),";
					break;

				case 4://case of integer
					create_query+=" int,";
					break;

				case 7://case of String
					create_query+=" float,";
					break;


				default:
					break;
			}

		}

		column = md.get(i);
		create_query+=column.columnName;

		switch (column.columnType) {
		case 1://case of vachar
		case 12:
			create_query+=" varchar(255))";
			break;

		case 4://case of integer
			create_query+=" int)";
			break;

		case 7://case of String
			create_query+=" float)";
			break;
```

```
            default:
                  break;
      }

            try {
                  Statement stmt1 = a.var.createStatement();
                  stmt1.execute (create_query);
                  stmt1.close();

            } catch (SQLException e1) {
                  System.err.println("[EasyQL
error][EQInterpreter.createTable]: create table failure");
                  //e1.printStackTrace();
            }
            try {
                  Statement stmt2 = a.var.createStatement();
                  ResultSet rs = stmt2.executeQuery(select_query);
                  ret = new EQTable (rs);

            } catch (SQLException e2) {
                  System.err.println("[EasyQL
error][EQInterpreter.createTable]: select failure");
                  //e2.printStackTrace();
            }

            return ret;
      }

      private void clearTable (EQTable a) {
            EQConn defConn =(EQConn) symt.getValue("DefConn");
            clearTable (defConn,a);
      }

      private void clearTable (EQConn a, EQTable b) {
            String query1 = "drop table "+b.name;

            try {
                  Statement stmt1 = a.var.createStatement();
                  stmt1.execute (query1);
                  stmt1.close();

            } ca  }
}
```

EQMySQLConn.java

```
import java.sql.*;

public class EQMySQLConn extends EQConn {

      public EQMySQLConn () {
            this.var = null;
      }

      public EQMySQLConn ( String hostname, String port, String dbname,
String id, String passwd) {
```

```
            try {
                  Class.forName("com.mysql.jdbc.Driver");
                  this.var =
DriverManager.getConnection("jdbc:mysql://69.22.220.234:"+port+"/"+dbname
,id,passwd);
            } catch (ClassNotFoundException e1) {
                  System.err.println("[EasyQL error][EQMySQLConn]: JDBC
loading failure");
                  System.exit(0);
                  //e1.printStackTrace();
            } catch (SQLException e2) {
                  System.err.println("[EasyQL error][EQMySQLConn]: Failed
to get connection");
                  System.exit(0);
                  //e2.printStackTrace();
            }
      }

      public EQTable listTable() {

            EQTable ret=null;

            try {
                  Statement stat = this.var.createStatement();
                  ResultSet rs = stat.executeQuery("show tables");
                  ret = new EQTable (rs);

            } catch (SQLException e1) {
                  System.err.println("[EasyQL
error][EQMySQLConn.listTable]: failed to run show table");
                  e1.printStackTrace();
            }
            return ret;
      }
}
```

EQSymTable.java

```
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Iterator;


public class EQSymTable extends HashMap {

      public final EQDataType getValue( String name) {
          Object x = this.get( name );
            return (EQDataType) x;
      }

      public final void setValue( String name, EQDataType data) {
            this.put( name, data);
      }

      public void what( PrintWriter output ) {
          for ( Iterator it = values().iterator() ; it.hasNext(); )
```

```
            {
                EQDataType d = ((EQDataType)(it.next()));
                d.what( output );
            }
      }

      public void what () {
            what(new PrintWriter( System.out, true ));
      }
}
```

EQTable.java

```
import java.io.PrintWriter;

import java.sql.*;

public class EQTable extends EQDataType {
      public ResultSet var;
      public EQTableMD md;

      public EQTable () {
            this.var = null;
      }

      public EQTable query(String attlist, String condlist) {
            try
            {
                  if(attlist.equals("")||(attlist==null))
                        attlist = " * ";
                  if(!(condlist.equals(""))&&(condlist!=null))
                        condlist = " WHERE "+condlist;

                  Statement stmt = var.getStatement();
                  String tab_name = var.getMetaData().getTableName(1);

                  ResultSet rs2;
                  String query = "SELECT " + attlist + " From " +
var.getMetaData().getTableName(1)
                                          + condlist;

                  rs2 =
stmt.getConnection().createStatement().executeQuery(query);

                  return new EQTable(rs2);
            }
            catch (SQLException e2)
            {

                  //Exception when executing java.sql related commands,
print error message to the console
                  System.err.println("[EasyQL error][EQTable.query]:
Failed to execute the select query");
                  //System.err.println(e2.toString());
            }

            return null;
      }
```

```
      public EQTable (ResultSet a) {
            this (null, a);
      }

      public EQTable (String a,ResultSet b) {
            this.name = a;
            this.var = b;
            this.md = getMetaData (this.var);
      }

      public String typename() {
        return "Table";
    }

      public EQTable copy (){
      return new EQTable (var);
    }

      public EQInt store(EQConn C) {
            Connection c= C.var;
            try
            {
                  EQTable a;
                  try
                  {
                        ResultSet rs=
c.createStatement().executeQuery("select * from
"+var.getMetaData().getTableName(1));
                        a = new EQTable(rs);
                  }
                  catch (SQLException e)
                  {
                        var.getMetaData().getTableName(1);
                        ResultSet rs = createTable (C,
var.getMetaData().getTableName(1));
                        a = new EQTable(rs);
                  }

                  //Check MD: by Kangkook
                  if (!this.md.equals (a.md)) {
                        System.out.println("SmridhinInsertTable3");
                        a.error("store operation fails - schema didn't
match");
                        return new EQInt (0);
                  }

                  insertTable(a.var);
                  return new EQInt(1);
            }
            catch (SQLException e)
            {
                  System.err.println("[EasyQL error][EQTable.store]:
store failure");
                  //System.out.println(e.getMessage());
            }
            return new EQInt(0);
      }
      private ResultSet createTable (EQConn a, String tname) {
```

```
            ResultSet ret=null;

            if (this.var==null || this.md==null)
                  this.error("creatTable error - ResultSet, Metadata not
initialized");

            EQTableMD md = this.md;
            int columnSize = md.size();
            EQColumn column = null;

            String select_query  = "select * from "+tname;
            String create_query  = "create table "+tname+ " (";
            int i;

            for (i=0;i<columnSize-1;i++) {

                  column = md.get(i);
                  create_query+=column.columnName;
                  switch (column.columnType) {
                        case 1://case of vachar
                        case 12:
                              create_query+=" varchar(255),";
                              break;

                        case 4://case of integer
                              create_query+=" int,";
                              break;

                        case 7://case of String
                              create_query+=" float,";
                              break;


                        default:
                              break;
                  }
            }

            column = md.get(i);
            create_query+=column.columnName;

            switch (column.columnType) {
            case 1://case of vachar
            case 12:
                  create_query+=" varchar(255))";
                  break;

            case 4://case of integer
                  create_query+=" int)";
                  break;

            case 7://case of String
                  create_query+=" float)";
                  break;

            default:
                  break;
      }
```

```
            try {
                  //System.out.println ("create table:"+create_query);
                  Statement stmt1 = a.var.createStatement();
                  stmt1.execute (create_query);
                  stmt1.close();

            } catch (SQLException e1) {
                  e1.printStackTrace();
            }
            try {
                  Statement stmt2 = a.var.createStatement();
                  ResultSet rs = stmt2.executeQuery(select_query);
                  ret = rs;
            } catch (SQLException e2) {
                  e2.printStackTrace();
            }
            return ret;
      }

      public EQTable insertTable(ResultSet a) {
            try
            {
                  //Insert tuple by tuple from VAR["TABLE_NAME"] to
Connection["TABLE_NAME"]: by Smridh
                  Statement stmt = a.getStatement();
                  String tab_name = a.getMetaData().getTableName(1);
                  int columnCount = var.getMetaData().getColumnCount();
                  var.beforeFirst();
                  int rowCount = 0;
                  while (var.next())
                 {
                     rowCount++;
                     Object o = null;
                     int i = 0;
                     StringBuffer insertValues = new StringBuffer();

                     for (i = 1; i < columnCount + 1; i++)
                     {
                         o = var.getObject(i);

                         String type =
var.getMetaData().getColumnTypeName(i);
                         surroundWithQuotes(type, insertValues, o);
                         if (i != columnCount)
                         {
                             insertValues.append(", ");
                         }
                     }

                       // Write in new table
                       a = stmt.executeQuery("SELECT * from "+tab_name);

                          //   conn.createStatement().
                       String query = "insert into "
                             +  tab_name + " values ("
                             + insertValues + ")";
                       stmt.executeUpdate(query);
             }
```

```
               var = stmt.executeQuery("SELECT * from "+tab_name);
            }
            catch (SQLException e)
            {
                  System.err.println("[EasyQL error][EQTable.insert]:
insert failure");
                  //System.out.println(e.getMessage());
            }

         return new EQTable(var);
      }

      private static void surroundWithQuotes(
            String type, StringBuffer sb, Object o)
        {
            if
(type.toUpperCase().equals("uniqueidentifier".toUpperCase()) ||
                type.toUpperCase().equals("string".toUpperCase()) ||
                type.toUpperCase().equals("varchar2".toUpperCase()) ||
                type.toUpperCase().equals("varchar".toUpperCase()) ||
                type.toUpperCase().equals("datetime".toUpperCase()))
            {
                if (o != null)
                {
                    sb.append("\'");
                    String string = o.toString();
                    String newString = string.replaceAll("\'", "\'\'");
                    sb.append(newString);
                    sb.append("\'");
                }
                else
                {
                    sb.append("null");
                }
            }
            else
            {
                sb.append(o);
            }
        }

      public EQInt delete(String condlist) {
            try
            {
                  String query = "DELETE From " +
var.getMetaData().getTableName(1)
                                          + " WHERE " + condlist;
                  Statement stmt = var.getStatement();
                  System.out.println (query);
                  String tab_name = var.getMetaData().getTableName(1);
                  int rows = stmt.executeUpdate(query);
                  var = stmt.executeQuery("Select * from "+tab_name);
                  return new EQInt(rows);
            }
            catch (SQLException e)
            {
                  System.err.println("[EasyQL error][EQTable.delete]:
delete failure"+e.getMessage());
                  //System.out.println(e.getMessage());
```

```
            }
            return new EQInt(0);
      }

      public EQInt insert(String attlist, String inslist) {
            try
            {
                  String query = "INSERT INTO " +
var.getMetaData().getTableName(1)
                                          + " ( " + attlist + " ) "
                                          + " VALUES ( " + inslist + " )
";

                  Statement stmt = var.getStatement();
                  String tab_name = var.getMetaData().getTableName(1);
                  int rows = stmt.executeUpdate(query);
                  var = stmt.executeQuery("Select * from "+tab_name);
                  return new EQInt(rows);
            }
            catch (SQLException e)
            {
                  System.err.println("[EasyQL error][EQTable.insert]:
insert failure");
                  System.out.println(e.getMessage());
            }
            return new EQInt(0);
      }

      public EQInt update(String attlist, String condlist) {
            try
            {
                  String query = "UPDATE " +
var.getMetaData().getTableName(1)
                                          + " SET " + attlist
                                          + " WHERE " + condlist;
                  Statement stmt = var.getStatement();
                  String tab_name = var.getMetaData().getTableName(1);
                  int rows = stmt.executeUpdate(query);
                  var = stmt.executeQuery("Select * from "+tab_name);
                  return new EQInt(rows);
            }
            catch (SQLException e)
            {
                  System.out.println(e.getMessage());
            }
            return new EQInt(0);
      }

      public EQInt drop() {
            try
            {
                  var.getStatement().executeUpdate("DROP TABLE "  +
var.getMetaData().getTableName(1));

                  return new EQInt(1);
            }
            catch (SQLException e)
            {
```

```
                  System.err.println("[EasyQL error][EQTable.drop]: drop
failure");
                  //System.out.println(e.getMessage());
            }
            return new EQInt(0);
      }

      public EQTable describe() {
            try
            {     //Critical to check
                  ResultSet columns =
var.getStatement().getConnection().getMetaData().getColumns(null, "%",
var.getMetaData().getTableName(1), "%");
                  return new EQTable(columns);
            }
            catch (SQLException e)
            {
                  System.err.println("[EasyQL error][EQTable.describe]:
describe failure");
                  //System.out.println(e.getMessage());
            }
            return null;
      }

      public EQInt create(EQConn C, String table_name, String
create_string) {
            Connection c= C.var;
            try
            {
                  String query = "Create table " + table_name
                                          + " ( " + create_string + " )
";
                  Statement stmt = c.createStatement();
                  String tab_name = table_name;
                  int rows = stmt.executeUpdate(query);
                  var = stmt.executeQuery("Select * from "+tab_name);
                  return new EQInt(1);
            }
            catch (SQLException e)
            {
                  //System.err.println("[EasyQL error][EQTable.create]:
create failure");
                  //e.printStackTrace();
            }
            return new EQInt(0);
      }

      public EQInt next (){
            try
            {
                  if(var.next())
                        return new EQInt(1);
            }
            catch (SQLException e2)
            {
                  //Exception when executing java.sql related commands,
print error message to the console
                  System.err.println("[EasyQL error][EQTable.next]: next
failure");
```

```
                  //System.err.println(e2.toString());
            }

      return new EQInt(0);
    }

      public EQInt reset (){
            try
            {
                  if(var.first())
                        return new EQInt(1);
            }
            catch (SQLException e2)
            {
                  //Exception when executing java.sql related commands,
print error message to the console
                  System.err.println("[EasyQL error][EQTable.reset]:
reset failure");
                  //System.err.println(e2.toString());
            }

      return new EQInt(0);
    }

      public EQDataType fetch (String attname){
            try
            {
                  var.next();
                  int columnCount = var.getMetaData().getColumnCount();

                  for (int i = 1; i < columnCount + 1; i++)
                  {
                String type = var.getMetaData().getColumnTypeName(i);
                String name = var.getMetaData().getColumnName(i);

                System.out.println (name);

                if(name.equals(attname))
                {
                    if
(type.toUpperCase().equals("INTEGER".toUpperCase()))
                    {
                          int a = var.getInt(attname);
                          return new EQInt(a);
                    }

                    if
((type.toUpperCase().equals("DOUBLE".toUpperCase()))

||(type.toUpperCase().equals("DECIMAL".toUpperCase()))

||(type.toUpperCase().equals("FLOAT".toUpperCase()))

||(type.toUpperCase().equals("REAL".toUpperCase())))
                    {
                          double a = var.getDouble(attname);
                          return new EQFloat(a);
                    }
```

```
                        if
((type.toUpperCase().equals("VARCHAR".toUpperCase()))

||(type.toUpperCase().equals("VARCHAR2".toUpperCase())))
                        {
                              String a = var.getString(attname);
                              return new EQVarchar(a);
                        }
                    }
                     }
              }
              catch (SQLException e2)
              {
                    //Exception when executing java.sql related commands,
print error message to the console
                    System.err.println("[EasyQL error][EQTable.fatch]:
fatch failure");
                    e2.printStackTrace();
                    System.exit(0);
                    //System.err.println(e2.toString());
              }

      return null;
    }

      public void print (PrintWriter w) {
            try
            {
                  if(var!=null) {
                        w.println("Table Name: " +
var.getMetaData().getTableName(1));

                        //Print Column Names:
                        ResultSetMetaData varMD = var.getMetaData();
                        for (int i = 0 ; i < varMD.getColumnCount () ;
i++)
                        {
                            int columnType = varMD.getColumnType (i + 1);
                            String columnName = varMD.getColumnLabel (i +
1);


                            w.print(columnName + "\t");
                            /*
                            ** Do something with columnType & columnName.
                            */

                        }
                        w.println("");

      w.println("====================================");

                        //Printg Column Values
                        int columnCount =
var.getMetaData().getColumnCount();

                        while(var.next()) {
                              for(int i=1; i<=columnCount; i++) {
                                          w.print(var.getString(i) +
"\t");
```

```
                              }
                              w.println("");
                        }
                  }
            }
            catch (SQLException e2)
            {
                  //Exception when executing java.sql related commands,
print error message to the console
                  System.err.println("[EasyQL error][EQTable.print]:
print failure");
                  //System.err.println(e2.toString());
            }
      }

      private EQTableMD getMetaData (ResultSet rs) {

            ResultSetMetaData rsmd;
            int numberOfColumns;

            EQTableMD ret = null;

            try {
                  rsmd = rs.getMetaData();
                  numberOfColumns = rsmd.getColumnCount();

                  ret = new EQTableMD ();

              for (int i=1; i<numberOfColumns+1;i++) {

                  EQColumn col = new EQColumn (rsmd.getColumnName(i),
rsmd.getColumnType(i));
                  ret.put(i-1,col);
              }
            } catch (SQLException e1) {
                  System.err.println("[EasyQL error][EQTable.next]: nexts
failure");
                  e1.printStackTrace();
            }
            return ret;
      }
}
```

EQTableMD.java

```
import java.util.HashMap;

public class EQTableMD extends HashMap<Integer, EQColumn> {
      String tableName;

      public EQTableMD () {
            this.tableName = null;
      }

      public EQTableMD (String tableName) {
            super();
            this.tableName = tableName;
      }
```

```
	public boolean equals (EQTableMD a) {
		//System.out.println("EQTableMD:1");
		boolean ret = true;

		for (int i=0;i<this.size();i++) {
		  if (!((EQColumn) this.get(i)).equals(a.get(i)))
			ret = false;
		}
		//System.out.println("EQTableMD:2"+":"+ret);
		return ret;
	}
}
```

EQVarchar.java

```
import java.io.PrintWriter;

public class EQVarchar extends EQDataType {
	public String var;

	public EQVarchar () {
		super ();
	}

	public EQVarchar (String a) {
		this.var =a;
	}

	public String typename() {
	  return "Varchar";
    }

    public static String varcharValue( EQDataType b ) {
        if ( b instanceof EQVarchar )
            return ((EQVarchar) b).var;
        return null;
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.println((String) var);
    }


    public EQVarchar copy(){
      return new EQVarchar (var);
    }
}
```

Main.java

```
import java.io.*;

import antlr.CommonAST;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;

public class Main {
  public static void main(String args[]) {

        if (args.length !=0)
            execFile (args[0]);
        else
            commandLine ();
  }

  public static void execFile(String filename){
          try {
          FileInputStream fileInput = null;
          fileInput = new FileInputStream(filename);


          DataInputStream input = new DataInputStream(fileInput);

            // Create the lexer and parser and feed them the input
            EASYQLLexer lexer = new EASYQLLexer(input);
            EASYQLParser parser = new EASYQLParser(lexer);
            parser.program(); // "file" is the main rule in the parser

            // Get the AST from the parser
            CommonAST parseTree = (CommonAST)parser.getAST();

            // Print the AST in a human-readable format
            System.out.println(parseTree.toStringList());

            //Printing out the tree walker result
            EASYQLTreeWalker treeParser = new EASYQLTreeWalker();

            // Open a window in which the AST is displayed graphically
            //ASTFrame frame = new ASTFrame("AST from the EasyQL parser",
parseTree);
            //frame.setVisible(true);

            EASYQLTreeWalker walker = new EASYQLTreeWalker ();
            EQDataType r = walker.expr (parseTree);

          } catch(Exception e) { System.err.println("Exception: "+e); }
}

  public static void commandLine () {
      InputStream input = (InputStream) new DataInputStream( System.in );
      //need walker here

      for ( ;; )
      {
          try
          {
              while( input.available() > 0 )
                  input.read();
          }
```

```
          catch ( IOException e ) {}

          System.out.print( "EasyQL> " );
          System.out.flush();

            EASYQLLexer lexer = new EASYQLLexer(input);
            EASYQLParser parser = new EASYQLParser(lexer);
            EASYQLTreeWalker walker = new EASYQLTreeWalker ();
            try {
              parser.cl_statement();
              CommonAST parseTree = (CommonAST)parser.getAST();
              System.out.println(parseTree.toStringList());
            } catch (Exception e) {}
       }
  }
}
```

EQColumn.java

```
public class EQColumn {
      String      columnName;
      int   columnType;
      String      columnTypeName;

      public EQColumn (String columnName, int colunmType) {
            this.columnName = columnName;
            this.columnType = colunmType;
      }

      public boolean equals (EQColumn a) {
            boolean ret = false;
            //if ((this.columnName.equals(a.columnName)) &&
(this.columnType == a.columnType))
            if (this.columnType == a.columnType) ret = true;
            return ret;
      }
}
```