EZQL Reference Manual

Bilal Bhatti
Syed Iqbal Ahmad

Introduction

This manual provides information for using the EZQL language for simplified database access. It is the standard for EZQL implementations, and any EZQL implementations must conform to this standard.

The style and syntax of EZQL is similar to Java, although vastly scaled down. EZQL files can be in an ASCII file. To run EZQL the target platform have an implementation of the java virtual machine. EZQL files are generated into java code.

Lexical Conventions

The first pass of the compiler through an EZQL will result in a set of tokens being produced.

Tokens

The significant tokens are Identifiers, Keywords, String literals, Operators and Constants. There is also white space which is ignored except to separate tokens, and comments which are ignored.

Whitespace

Whitespace can be ' ', '\r', '\n', \t'. They can separate tokens, such as:

int x;

There is a space between int and x. There could be one space or fifty spaces.   (any number actually, fifty is just an example).

Lexical analysis continues to the end of the file.

Comments

Only single line comments are allowed. They begin with // and continue until and valid newline character is found.

*COMMENTS : "//" (~('\n'|'\r'))**

Identifier

An identifier can consist of letters, digits and underscores. It must begin with a letter. They can have any length. They are case-sensitive (The identifier 'num' is different than 'NUM').

*ID : ('a'..'z' |'A'..'Z') ('a'..'z' | A'..'Z' | '0'..'9' | '_') *;*

Keywords

All keywords that are specified in the Java specification are reserved. They may or may not be used in EZQL.
Further we introduce some new keywords:

as
list
query
row
table
view

Constants

There are a variety of constant types. They are integers, floating point and boolean.

Integers are any valid base 10 numbers that consist of a sequence of digits.
Example: 123

Floats must have an integer part, a decimal point and a value after the decimal.
Example: 0.0

Boolean constants are only "true" or "false".

String Literals

A string is a sequence of characters enclosed in " ".

Operators

Boolean Operators: &&, ||, !
Arithmetic Operators: +, -, *, /, %
Logical Operators: <=, >=, >, <

Rules of precedence are the same as expected for languages in common use.
For math operators *, /, % have more precedence then +,-.
Parenthesis will have more precedence.

Boolean operators and logical operators have precedence left to right.

Program Structure

The program structure is as follows.

```
compilationUnit
        : (packageDeclaration)?
          (tableDefinition)
```

A compilation unit is a valid EZQL program. It consists of an optional package declaration and a table definition.
The table definition will consist of smaller portions which are needed to make an EZQL program.

```
packageDeclaration
        : "package" ID (DOT ID)* SEMI
        ;
```

ID refers to Identifier, DOT refers to a single period and SEMI refers to a single semicolon.

```
tableDefinition
        : "table" ID "as" ID LCURLY
            (methodDefinition)*
          RCURLY
        ;
```

LCURLY and RCURLY are { and } respectively.

A table represents a single file. The first ID after table must match the name of the file. The second ID will be used in the generated code.

```
methodDefinition
        : ("row"|"list"|"void") ID LPAREN (argumentList)? RPAREN LCURLY
            (statement)*
          RCURLY
        ;
```

A method definition begins by indicating the amount of data the query is expected to return, either a single row or a list.

Further a method takes arguments. They are as defined below:

```
argumentList
        : argumentSpec (COMMA argumentSpec)*
        ;
```

```
argumentSpec
        : ("int"|"string"|"boolean") ID
```

This definition for arguments is basically a comma-separated list of arguments, such as:

int x, int y

Statements

Statements are indicated in method definition. They are executed in order.

Valid Statements are:

```
assignmentStatement
        : ID ASSIGN expression SEMI
        ;
```

Assign is an =
Expression will be explained in more detail below.

```
returnStatement
        : "return" (ID)? SEMI
        ;
```

ID in return statement must be of type query.

```
whileStatement
        : "while" LPAREN relationalExpression RPAREN LCURLY
                (statement)*
          RCURLY
        ;
```

RelationalExpression is simply an expression, which evaluates to true or false.
It will be explained in more detail below.

```
ifStatement
        : "if" LPAREN relationalExpression RPAREN LCURLY
                (statement)*
          RCURLY
        ;
```

Declarations

They consist of
decl : type ID (init)?

Valid types are int, string, boolean, query.
The optional init block is for initialization, such as:

int x = 0;

Declared variables have a scope of the nearest { }.

Expressions

Expression elements and constant values are the smallest portion of an expression.

```
expressionElements
        : constantValues
        | LPAREN expression RPAREN
        ;

constantValues
        : INTEGER
        | CHAR_LITERAL
        ;
```

Expressions follow precedence rules. The expressions are listed below in precedence order.

Further they are evaluated left to right for any rules at the same level.

```
negationExpression
        : (LNOT)? expressionElements
        ;

signExpression
        : (PLUS | MINUS)? negationExpression
        ;

multiplyingExpression
        : signExpression ((STAR | DIV | MOD) signExpression)*
        ;
```

```
additionExpression
        : multiplyingExpression ((PLUS | MINUS) multiplyingExpression)*
        ;

relationalExpression
        : additionExpression ((EQUAL | NOT_EQUAL | LT | LE | GT | GE)
additionExpression)*
        ;

andExpression
        : relationalExpression (LAND relationalExpression)*
        ;
```