University of Southampton

Programming Language Concepts

COMP2212


# Mapex Query Language - User Manual

Ori Lazar
Isaac Edmonds

May 2018

# Contents

# 1 Introduction

Mapex Query Language (MQL) was originally created with the goal of being able to write a program to solve any particular conjunctive query. Our main inspiration came from SQL as we wanted to create a simple human readable programming language which can be taught to non-programmers as well as veteran programmers.

# 2 Language Specification (Syntax)

## 2.1 BNF

The BNF gives a formal representation of MQL's context-free grammar, it can be found in [B.3].

## 2.2 Lexical Grammar

MQL's grammar's terminal symbols are made from the Unicode character set. It's grammar defines a set of productions, starting from the goal symbol { , that describe how sequences of Unicode characters are translated into a sequence of tokens [2.2.6]. The lexing stage ignores whitespace [2.2.2] and comments [2.2.4]. The resulting tokens are the identifiers [2.2.5], keywords and separators [2.2.6] and are passed through to happy parser file.

### 2.2.1 Line Terminators

Line terminators, **';'** , support the different conventions of existing languages (e.g Java). These separators denote the end of a statement and can be staggered to write code on multiple lines (given a matching format).

### 2.2.2 White Space

White space is defined as the ASCII space character, horizontal tab character, form feed character and line terminator characters, all of which are ignored during the lexing phase.

### 2.2.3 NULL Character

MQL uses a null character '␀' (Unicode value:U+2400) to denote empty spaces within its tables. These are then cleared during printing. The use of ␀has been restricted so that an exception is thrown when it exists in the read file (empty space is accepted). Insert statements can be used to add '␀' elements with the character as-well.

### 2.2.4 Comments

The commenting structure in MQL uses end-of-line comments, denoted by two **'-'** characters. All the text on the rest of the line is then ignored. We also have introduced multi-line block commenting with '/*' to start a comment and '*/' to end, however, this is only within the program borders ('{' and '}').

### 2.2.5 Identifiers

An identifier is an unlimited-length sequence of Unicode characters (minus token strings), which are used for referencing table & column names, as well as comparing columns to literal strings. This allows programmers to use identifiers in their programs that are written in their native languages.

### 2.2.6 Tokens: Keywords and Separators

Tokens in MQL are the combination of all the unique sequences of ASCII characters used as keywords and separators. 32 character sequences, formed from ASCII characters, are reserved for use as keywords and separators, they therefore cannot be used as identifiers [B.1]. 7 tokens, 2 of which are grouped, are used as separators [B.2].

## 2.3 Execution

The sequence of execution of an MQL program is controlled by statements. Some statements in MQL contain other statements as part of their structure. Programs are defined as blocks (sequence of statements within braces). A block is executed by executing statements in order from first to last. If all of these block statements complete normally then so does the block. For this case in MQL programs start with **'{'** and terminate with **'}'**.

### 2.3.1 Quotes and Brackets

In MQL brackets are used to denote associativity, therefore brackets can be added or removed from any statement. Brackets can also be removed from anywhere in the statement below. Quotes are used in MQL to denote strings/-values, used only in some of the statements below (CREATE,INSERT,READ,DUPLICATE), and must be used for the statements to parse correctly.

### 2.3.2 Table Declaration Statement

A table declaration statement instantiates a new table. An error will be thrown if a table with that name already exists within the program state.

```
CREATE TableName ("Coulmn1","Column2");
```

### 2.3.3 Drop Table Statement

A table drop statement removes an already existing table, throwing an error if that table doesn't exist in the state.

```
DROP TABLE TableName;
```

### 2.3.4 Insert Values Into Table Statement

An insert statement adds values to an already existing table, throwing an error if the table doesn't exist, or, if the number of columns does not match the table. All columns for a table must be inserted into and empty values can be inserted with 'NUL' [2.2.3].

```
INSERT INTO TableName (TableName.Coulmn1,TableName.Column2) VALUES ("Value1","Value2");
```

### 2.3.5 Read File Into Table Statement

A read statement will parse through a .csv file, inserting its values into a named table. Errors are thrown if: the table doesn't exists; the file doesn't exist; unsupported characters are used (mainly tokens). Tables can have multiple read statements into them as it is just inserting values.

```
READ "FileName" INTO TableName;
```

### 2.3.6 Duplicate Table Statement

A duplicate statement will duplicate a table into a different table, throwing a errors for non-existing or non-matching tables (different number of columns).

```
DUPLICATE "TableName" INTO TableName;
```

### 2.3.7 Select Statements

The SELECT statement returns a result set of records from one or more tables. The select statement may be used in combination with one of multiple clauses, these being join, where and order clauses. For simplicity within the domain of conjunctive queries only one select statement can be used per program where the program will terminate.

#### 2.3.7.1 Base Select Feature

In a select statement the user can either select individual columns by using the (A.Column1,A.Column2) syntax or by selecting all the columns with the '*' syntax. Within the listing syntax any number of columns can be selected, however, they must be present in the result table to be valid. An error is thrown if no select statement is parsed. Below is an example of a simple select query.

```
SELECT Table1.Column1 FROM (Table1);
SELECT (Table1.Column1, Table1.Column2) FROM (Table1);
SELECT * FROM (Table1);
```

#### 2.3.7.2 Join Clause

The JOIN clause joins the values from two (or more) tables, causing each element in A to match with each element in B (similar to other QL joins [SQL]). A syntactic sugar of (A,B,C) can also be used. Joins can be stacked, however, you cannot combine the two types of join syntax.

```
SELECT (Table1.Column1) FROM (Table1 JOIN Table2);
SELECT (Table3.Column1) FROM (Table1,Table2,Table3);
```

#### 2.3.7.3 Where Clause

The WHERE clause is used to filter records based on specific relational conditions. In MQL there are checks for 'EQUAL' and 'NOT EQUAL' with both being able to match column values and strings. Conditional logic can be used, where both 'AND' and 'OR' can be written in the where clause (Stacking these is possible).

```
SELECT (Table1.Column1) FROM (Table1) WHERE (Table1.Column1 EQUAL "Hello Worlds!");
SELECT (Table1.Column1) FROM (Table1) WHERE (Table1.Column1 NOT EQUAL Table1.Column2) AND
(Table1.Column1 EQUAL "Pawel");
SELECT (Table1.Column1) FROM (Table1) WHERE ((Table1.Column1 EQUAL Table1.Column2) AND (Ta-
ble1.Column1 NOT EQUAL "Julian")) OR (Table1.Column1 EQUAL "Pawel");
```

#### 2.3.7.4 Order By Clause

The ORDER BY clause is used to sort the results based on either a single column, or, all columns (using the '*' syntax). The direction of the sort can also be changed by using the 'ASCEND' and 'DESCEND' keywords to denote the direction.

```
SELECT (Table1.Column1) FROM (Table1) ORDER Table1.Column1 BY ASCEND;
SELECT (Table1.Column1,Table1.Column2) FROM (Table1) ORDER (Table1.Column1) BY (DESCEND);
SELECT (Table1.Column1) FROM (Table1) ORDER * BY ASCEND;
```

#### 2.3.7.5 Combining the select features

Below is an example of how all of the above select features can be combined to perform a query that sorts selected data based on some conditions.

```
SELECT * FROM (Table1 JOIN Table2) WHERE (Table1.Column1 EQUAL Table2.Column2) ORDER * BY
ASCEND;
```

# 3 Additional Features

## 3.1 Syntax Error Handling

MQL provides the programmer with informative syntax error messages which include the statement in which the error occurred and a response of how to potentially fix this error.

## 3.2 Syntax Highlighting

MQL provides an optional syntax highlighting feature, based in Notepad++. This was created for MQL and is shown in the problem screenshots in the appendix. As well as this MQL also shares much of its syntactic structure with other query languages, such as SQL, thus their syntax highlighting can be used for MQL in popular IDE solutions.

## 3.3 Syntactic Sugars

MQL provides 'syntactic sugars', which are alternative ways of defining the same actions, each of which have their own benefits.

**Join or A,B:** The JOIN condition in select statement [2.3.7.2] has syntactic sugar to either be defined statically with (A JOIN B) or with a shorter syntax of (A,B). Both of these methods are valid and provide no system benefits besides their appearance.

**Select '*' Syntax:** Select statements in MQL can be long and tedious to write out , we have therefore included a shortcut syntax to select all columns of the result table [2.3.7]. This syntax, denoted by '*', also has some performance benefits other than just its neater appearance. It speeds up the select statement by skipping the check for if all of the inputted select columns exist in the results table, however, this performance benefit has negligible impact.

**Duplicate Table Statement:** MQL provides a statement called DUPLICATE [2.3.6], this statement will duplicate any given table into another. The same functionality can be done by reading and joining tables. This feature has sizable performance benefits due to its simplicity of copying the table (rather than reading one from a file).

## 3.4 Comments

Comments are included in MQL to give the user the ability to describe and annotate their code. Information on the two types of comments can be found in the grammar section [2.2.4].

## 3.5 Direct String Comparison & OR Conditional Logic

MQL focuses on solving conjunctive queries, however, some extra features were added to enable the user to use it for a wider range of applications. One of these features was the ability to compare table values against strings [2.3.7.3] which enables the user to test and filter their output even further. Another extra feature is the use of OR conditional logic for the same reasons as above [2.3.7.3].

## 3.6 Order by Descending

Ordering/Sorting by column value is a feature MQL uses for lexicographic sorting (ascending order), as well as this, MQL also has the functionality for a descending order sort [2.3.7.4].

## 3.7 Insert Values & Drop Table Statements

During the development of MQL, testing was required, therefore, insert [2.3.4] and drop table [2.3.3] statements were added for to enhance the usability of the system by allowing users to enter specific values, as opposed to reading them from a file.

# A Programs 1 - 10

## A.1 Program - 1 - Conjunction

$$x1, x3, x2, x4 \vdash A(x1, x2) \wedge B(x3, x4) \tag{1}$$

```
-- pr1 performs the conjunctive query:   x1, x3, x2, x4 |- A(x1, x2) ∧ B(x3, x4)
{
  /*
    The sequence of pr1.cql is:
      [1] We create 2 tables with 2 columns.
      [2] We read in .csv file A into table A.
      [3] We duplicate table A into table B.
      [4] We select the columns based on what was requested in the query
          from the join of the two tables and then the result is outputted to the terminal
  */
  CREATE TABLE A("Column1","Column2");
  CREATE TABLE B("Column1","Column2");
  READ "A" INTO A;
  READ "B" INTO B;
  SELECT (A.Column1, B.Column1, A.Column2, B.Column2) FROM (A JOIN B) ORDER * BY ASCEND;
}
```

## A.2 Program - 2 - Conjunction and variable repetition

$$x1, x3, x2 \vdash A(x1, x2) \wedge B(x2, x3) \tag{2}$$

```
-- pr2 performs the conjunctive query:   x1, x2, x3 |- A(x1, x2) ∧ B(x2, x3)
{
  /*
    The sequence of pr2.cql is:
      [1] We create 2 tables with 2 columns.
      [2] We read in .csv file A into table A.
      [3] We duplicate table A into table B.
      [4] We select the columns based on what was requested in the query from the join of the two tables
          where the condition requested in the query is met.
  */
  CREATE TABLE A("Column1","Column2");
  CREATE TABLE B("Column1","Column2");
  READ "A" INTO A;
  READ "B" INTO B;
  SELECT (A.Column1, B.Column1, B.Column2) FROM (A,B) WHERE (A.Column2 EQUAL B.Column1) ORDER * BY ASCEND;  -- (A,B) is Syntactic Sugar for A JOIN B.
}
```

### A.3 Program - 3 - Equality

$$x1, x2 \vdash P(x1) \vdash Q(x2) \vdash x1 = x2 \tag{3}$$

```
-- pr3 performs the conjunctive query:   x1, x2 |- P(x1) ∧ Q(x2) ∧ x1 = x2
{
  /*
    The sequence of pr3.cql is:
      [1] We create 2 tables with 1 column each.
      [2] We read in .csv file P into table P.
      [3] We read in .csv file Q into table Q.
      [4] We select the columns based on the join of the two tables and order by ascending order
  */
  CREATE TABLE P("Column1");
  CREATE TABLE Q("Column1");
  READ "P" INTO P;
  READ "Q" INTO Q;
  SELECT (P.Column1, Q.Column1) FROM (P JOIN Q) WHERE (P.Column1 EQUAL Q.Column1) ORDER * BY ASCEND;
}
```

### A.4 Program - 4 - Existential quantification

$$x1 \vdash \exists z.R(x1, z) \tag{4}$$

```
-- pr4 performs the conjunctive query:   x1 |- ∃z.R(x1, z)
{
  /*
    The sequence of pr4.cql is:
      [1] We create a table with 2 columns.
      [2] We read in .csv file R into table R.
      [3] We select the column from the table R based on ascending order.
  */
  CREATE TABLE R("Column1", "Column2");
  READ "R" INTO R;
  SELECT (R.Column1) FROM (R) ORDER * BY ASCEND;
}
```

### A.5 Program - 5 - Existential quantification and conjunction

$$x1, x2 \vdash \exists z.A(x1, z) \wedge B(z, x2) \tag{5}$$

```
-- pr5 performs the conjunctive query:   x1, x2 |- ∃z.A(x1, z) ∧ B(z, x2)
{
  /*
    The sequence of pr5.cql is:
      [1] We create 2 tables with 2 columns A and B.
      [2] We read in two .csv files into the respective tables.
      [3] We then select with the condition of EQUAL column values and order.
  */
  CREATE TABLE A("Column1","Column2");
  CREATE TABLE B("Column1","Column2");
  READ "A" INTO A;
  READ "B" INTO B;
  SELECT (A.Column1, B.Column2) FROM (A JOIN B) WHERE (A.Column2 EQUAL B.Column1) ORDER * BY ASCEND;
}
```

### A.6 Program - 6 - Check for emptiness

$$x1 \vdash R(x1) \wedge \exists z.S(z) \tag{6}$$

```
-- pr6 performs the conjunctive query: x1 |- R(x1) ∧ ∃z.S(z)
{
  /*
    The sequence of pr6.cql is:
      [1] We create 2 tables with a single column.
      [2] We read in .csv file R into table R.
      [3] We read in .csv file S into table S.
      [4] We select the column from table R from the join of the two tables, this is then output in ascending order.
  */
  CREATE TABLE R("Column1");
  CREATE TABLE S("Column1");
  READ "R" INTO R;
  READ "S" INTO S;
  SELECT (R.Column1) FROM (R JOIN S) ORDER * BY ASCEND;
}
```

### A.7 Program - 7 - Paths of length three

$$x1, x2 \vdash \exists z1.\exists z2.R(x1, z1) \wedge R(z1, z2) \wedge R(z2, x2) \tag{7}$$

```
-- pr7 performs the conjunctive query:    x1, x2 |- ∃z1.∃z2. R(x1, z1) ∧ R(z1, z2) ∧ R(z2, x2)
{
/*
  The sequence of pr7.cql is:
    [1] We create 3 tables with 2 columns.
    [2] We read in .csv file A into table A.
    [3] We duplicate table A into table B.
    [4] We duplicate table A into table C.
    [5] We then select with the condition of EQUAL column values and order.
  */
  CREATE TABLE A("Column1", "Column2");
  CREATE TABLE B("Column1", "Column2");
  CREATE TABLE C("Column1", "Column2");
  READ "R" INTO A;
  DUPLICATE "A" INTO B; -- Syntactic sugar for READ R INTO B;
  DUPLICATE "B" INTO C; -- Syntactic sugar for READ R INTO C;
  SELECT (A.Column1,C.Column2) FROM (A JOIN B JOIN C) WHERE (A.Column2 EQUAL B.Column1 ) AND (B.Column2 EQUAL C.Column1) ORDER * BY ASCEND;
}
```

### A.8 Program - 8 - Cycles of length 4

$$x1, x2, x3, x4, x5 \vdash R(x1, x2) \wedge R(x2, x3) \wedge R(x3, x4) \wedge R(x4, x5) \wedge x1 = x5 \tag{8}$$

```
-- pr8 performs the conjunctive query:    x1, x2, x3, x4, x5 |- R(x1, x2) ∧ R(x2, x3) ∧ R(x3, x4) ∧ R(x4, x5) ∧ x1 = x5
{
  /*
    The sequence of pr8.cql is:
      [1] We create 4 tables with 2 columns each to join 4 times with file R.
      [2] We read in .csv file R into table R and then duplicate this table into the 3 other tables.
      [3] We then select the required rows that match the query above, getting the columns for the required values.
      [4] We then perform all of the conditional checks required for the cycle check.
  */
  CREATE TABLE R("Column1", "Column2");
  CREATE TABLE S("Column1", "Column2");
  CREATE TABLE T("Column1", "Column2");
  CREATE TABLE U("Column1", "Column2");

  READ "R" INTO R;
  DUPLICATE "R" INTO S; --Syntactic sugar for READ "R" INTO S
  DUPLICATE "R" INTO T; --Syntactic sugar for READ "R" INTO T
  DUPLICATE "R" INTO U; --Syntactic sugar for READ "R" INTO U
  SELECT (R.Column1,R.Column2,T.Column1,T.Column2,U.Column2) FROM (R JOIN S JOIN T JOIN U)
  WHERE (R.Column2 EQUAL S.Column1) AND (S.Column2 EQUAL T.Column1) AND (T.Column2 EQUAL U.Column1) AND (R.Column1 EQUAL U.Column2) ORDER * BY ASCEND;
}
```

### A.9 Program - 9 - Triple composition

$$x1, x2 \vdash \exists z1.\exists z2.A(x1, z1) \wedge B(z1, z2) \wedge C(z2, x2) \tag{9}$$

```
-- pr9 performs the conjunctive query:    x1, x2 |- ∃z1.∃z2. A(x1, z1) ∧ B(z1, z2) ∧ C(z2, x2)
{
  /*
    The sequence of pr9.cql is:
      [1] We create 3 tables with 2 columns to insert the relevant files into.
      [2] After this we read the files into their respective tables.
      [3] We then select the required rows that match the query above, getting the columns for the required values.
  */
  CREATE TABLE A("Column1","Column2");
  CREATE TABLE B("Column1","Column2");
  CREATE TABLE C("Column1","Column2");
  READ "A" INTO A;
  READ "B" INTO B;
  READ "C" INTO C;
  SELECT (A.Column1,C.Column2) FROM (A JOIN B JOIN C) WHERE (A.Column2 EQUAL B.Column1) AND (B.Column2 EQUAL C.Column1) ORDER * BY ASCEND;
}
```

### A.10 Program - 10 - Check for pairs

$$x1, x2, x3 \vdash S(x1, x2, x3) \wedge (\exists z1.\exists z2.S(z1, z1, z2)) \wedge (\exists z1.\exists z2.S(z1, z2, z2)) \tag{10}$$

```
-- pr10 performs the conjunctive query:    x1, x2, x3 |- S(x1, x2, x3) ∧ (∃z1.∃z2. S(z1, z1, z2)) ∧ (∃z1.∃z2. S(z1, z2, z2))
{
  /*
    The sequence of pr10.cql is:
      [1] We create 3 tables with 3 columns to insert the respective files into.
      [2] After this we read file S into table R and then duplicate table R into tables S and T.
      [3] We then select the required rows that match the query above, getting the columns for the required values.
  */
  CREATE TABLE R("Column1","Column2","Column3");
  CREATE TABLE S("Column1","Column2","Column3");
  CREATE TABLE T("Column1","Column2","Column3");
  READ "S" INTO R;
  DUPLICATE "R" INTO S; -- Syntactic sugar for READ "S" INTO S;
  DUPLICATE "R" INTO T; -- Syntactic sugar for READ "S" INTO T;
  SELECT (R.Column1,R.Column2,R.Column3) FROM (R,S,T) WHERE (S.Column1 EQUAL S.Column2 AND T.Column2 EQUAL T.Column3) ORDER * BY ASCEND;
  -- (A,B) is Syntactic Sugar for A JOIN B.
}
```

# B References

## B.1 Token Keywords

- SELECT
- FROM
- JOIN
- WHERE
- INSERT
- VALUES
- ORDER
- BY
- CREATE
- DROP
- TABLE
- READ
- INTO
- DUPLICATE
- EQUAL
- AND
- OR
- NOT
- EQUAL
- ASCEND
- DESCEND
- *
- /*
- */

## B.2 Separator Keywords

- ( ) - Used to separate joins, conditions, lists of string and strings.
- { } - Used to separate blocks of code in a single program. See 2.3
- . - Used to separate a table and it's column "TableOne.ColumnOne"
- , - Used to separate the table names when selecting from multiple tables in joins.
- ; - used to separate different lines from each other.

## B.3 BNF Syntax

```
-------------------------------------
Mapex Query Language - Advisory BNF:
-------------------------------------

<StartProgram> ::= { <Program> }

<Program> ::= <Statement> <Program>
            | <Statement>

<Statement> ::=  /* <InfiniteComment> */
              | CREATE TABLE <String> <StringList> ;
              | READ " <String> " INTO <String> ;
              | DUPLICATE " <String> " INTO <String> ;
              | INSERT INTO <String> <SelectString> VALUES <StringList> ;
              | ( <Statement> )
              | <SelectStatement>

<InfiniteComment> ::= <InfiniteComment> <InfiniteString>
                    | <InfiniteString>

<InfiniteString> ::= <String> | . | ; | " | * | ( | ) | , | { | } | SELECT | FROM | JOIN
                   | WHERE | INSERT | VALUES | ORDER | BY | CREATE TABLE | DROP TABLE
                   | DUPLICATE | READ | INTO | EQUAL | AND | OR | NOT EQUAL | ASCEND | DESCEND


<SelectStatement> ::=  SELECT <SelectString> FROM <FromStatement> ;
                     | SELECT <SelectString> FROM <FromStatement> ORDER <OrderString> BY <Order> ;
                     | SELECT <SelectString> FROM <FromStatement> WHERE <Cnd> ;
                     | SELECT <SelectString> FROM <FromStatement> WHERE <Cnd> ORDER <OrderString> BY <Order> ;

<StringList> ::=  <StringList> , <StringList>
                | ( <StringList> )
                | " <String> "

<SelectString> ::= <SelectString> , <SelectString>
                 | ( <SelectString> )
                 | *
                 | <String> . <String>

<FromStatement> ::= <FromStatement> JOIN <FromStatement>
                  | ( <FromStatement> )
                  | <FromString>

<FromString> ::= <FromString> , <FromString>
               | <String>

<OrderString> ::=   *
                  | <String> . <String>
                  | ( <OrderString> )

<Order> ::= ASCEND
          | DESCEND
          | ( <Order> )

<Cnd> ::= <CndString> NOT EQUAL <CndString>
        | <CndString> EQUAL <CndString>
        | <Cnd> AND <Cnd>
        | <Cnd> OR <Cnd>
        | ( <Cnd> )

CndString ::= ( <CndString> )
            | " <String> "
            | <String> . <String>

<String> ::= <Character> <String>

<Character> ::= <letter> | <digit> | <symbol>

<letter>    ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K"
              | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V"
              | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g"
              | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"
              | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

<digit>     ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<symbol>    ::=  <UnicodeSymbol>? except " ( ) { } . , ; * / ~

? = Any unicode symbol referenced in UTF-8.

Single line comments and whitespace are not included in our BNF, this is due to their scope being outside of the
grammar definition. Where all white space is ignored and comments, "--", are ignored until the end of the line.

```

## B.4 Syntax Highlighting XML file

```
<NotepadPlus>
    <UserLang name="MQL" ext="" udlVersion="2.1">
        <Settings>
            <Global caseIgnored="no" allowFoldOfComments="no" foldCompact="no" forcePureLC="0"
                decimalSeparator="0" />
            <Prefix Keywords1="no" Keywords2="no" Keywords3="no" Keywords4="no" Keywords5="yes"
                Keywords6="no" Keywords7="no" Keywords8="no" />
        </Settings>
        <KeywordLists>
            <Keywords name="Comments">00-- 01 02 03/* 04*/</Keywords>
            <Keywords name="Numbers, prefix1"></Keywords>
            <Keywords name="Numbers, prefix2"></Keywords>
            <Keywords name="Numbers, extras1"></Keywords>
            <Keywords name="Numbers, extras2"></Keywords>
            <Keywords name="Numbers, suffix1"></Keywords>
            <Keywords name="Numbers, suffix2"></Keywords>
            <Keywords name="Numbers, range"></Keywords>
            <Keywords name="Operators1">, ; .</Keywords>
            <Keywords name="Operators2"></Keywords>
            <Keywords name="Folders in code1, open"></Keywords>
            <Keywords name="Folders in code1, middle"></Keywords>
            <Keywords name="Folders in code1, close"></Keywords>
            <Keywords name="Folders in code2, open"></Keywords>
            <Keywords name="Folders in code2, middle"></Keywords>
            <Keywords name="Folders in code2, close"></Keywords>
            <Keywords name="Folders in comment, open"></Keywords>
            <Keywords name="Folders in comment, middle"></Keywords>
            <Keywords name="Folders in comment, close"></Keywords>
            <Keywords name="Keywords1">CREATE TABLE READ SELECT FROM INTO WHERE AND DUPLICATE ORDER BY
                VALUES DROP AND OR </Keywords>
            <Keywords name="Keywords2"> *</Keywords>
            <Keywords name="Keywords3">JOIN </Keywords>
            <Keywords name="Keywords4">ASCEND DESCEND EQUAL NOT</Keywords>
            <Keywords name="Keywords5"></Keywords>
            <Keywords name="Keywords6"></Keywords>
            <Keywords name="Keywords7"></Keywords>
            <Keywords name="Keywords8"></Keywords>
            <Keywords name="Delimiters">00&amp;quot; 01 02&amp;quot; 03 04 05 06 07 08 09 10 11 12 13 14 15 16
                17 18 19 20 21 22 23</Keywords>
        </KeywordLists>
        <Styles>
            <WordsStyle name="DEFAULT" fgColor="000000" bgColor="FFFFFF" fontName="Courier New"
                fontStyle="0" nesting="0" />
            <WordsStyle name="COMMENTS" fgColor="808080" bgColor="FFFFFF" fontStyle="0" nesting="0" />
            <WordsStyle name="LINE COMMENTS" fgColor="595959" bgColor="FFFFFF" fontStyle="0" nesting="0
                " />
            <WordsStyle name="NUMBERS" fgColor="000000" bgColor="FFFFFF" fontStyle="0" nesting="0" />
            <WordsStyle name="KEYWORDS1" fgColor="4040BF" bgColor="FFFFFF" fontName="Courier New"
                fontStyle="1" nesting="0" />
            <WordsStyle name="KEYWORDS2" fgColor="B871FF" bgColor="FFFFFF" fontStyle="5" nesting="0" />
            <WordsStyle name="KEYWORDS3" fgColor="FF0000" bgColor="FFFFFF" fontStyle="0" nesting="0" />
            <WordsStyle name="KEYWORDS4" fgColor="FF8000" bgColor="FFFFFF" fontStyle="0" nesting="0" />
            <WordsStyle name="KEYWORDS5" fgColor="F0BB04" bgColor="FFFFFF" fontStyle="0" nesting="0" />
            <WordsStyle name="KEYWORDS6" fgColor="000000" bgColor="FFFFFF" fontStyle="0" nesting="0" />
            <WordsStyle name="KEYWORDS7" fgColor="000000" bgColor="FFFFFF" fontStyle="0" nesting="0" />
            <WordsStyle name="KEYWORDS8" fgColor="000000" bgColor="FFFFFF" fontStyle="0" nesting="0" />
            <WordsStyle name="OPERATORS" fgColor="FF8306" bgColor="FFFFFF" fontStyle="0" nesting="0" />
            <WordsStyle name="FOLDER IN CODE1" fgColor="000000" bgColor="FFFFFF" fontStyle="0" nesting=
                "0" />
            <WordsStyle name="FOLDER IN CODE2" fgColor="000000" bgColor="FFFFFF" fontStyle="0" nesting=
                "0" />
            <WordsStyle name="FOLDER IN COMMENT" fgColor="000000" bgColor="FFFFFF" fontStyle="0"
                nesting="0" />
            <WordsStyle name="DELIMITERS1" fgColor="52A03F" bgColor="FFFFFF" fontStyle="0" nesting="0"
                />
            <WordsStyle name="DELIMITERS2" fgColor="000000" bgColor="FFFFFF" fontStyle="0" nesting="0"
                />
            <WordsStyle name="DELIMITERS3" fgColor="000000" bgColor="FFFFFF" fontStyle="0" nesting="0"
                />
            <WordsStyle name="DELIMITERS4" fgColor="000000" bgColor="FFFFFF" fontStyle="0" nesting="0"
                />
            <WordsStyle name="DELIMITERS5" fgColor="000000" bgColor="FFFFFF" fontStyle="0" nesting="0"
                />
            <WordsStyle name="DELIMITERS6" fgColor="000000" bgColor="FFFFFF" fontStyle="0" nesting="0"
                />
            <WordsStyle name="DELIMITERS7" fgColor="000000" bgColor="FFFFFF" fontStyle="0" nesting="0"
                />
            <WordsStyle name="DELIMITERS8" fgColor="000000" bgColor="FFFFFF" fontStyle="0" nesting="0"
                />
        </Styles>
    </UserLang>
</NotepadPlus>
```