MX

Language Reference Manual

By: Aaron Jackson (arj2145), Wilderness Oberman (wo2168),
Rashel Rojas (rdr2139), Mauricio Guerrero (mg4145)

1. Introduction

Our proposed language, MX, aims to offer programmers an intuitive and efficient means of creating and manipulating matrices.

Although matrices are robust and powerful mathematical structures that are paramount to various fields of Computer Science - attempting to navigate them often results in unnecessary complexities. Moreover, most typical programming languages lack the coherent means of handling matrices without the additional importation of an outside library of some sort. Thus, MX seeks to make matrix processing all the more simpler through providing a streamlined experience of maneuvering matrices.

MX seeks to overhaul the current matrix handling experience by providing one that should be both intuitive and familiar to programmers. MX aims to be intuitive to programmers through its inclusion of the matrix as a data type. By doing this, it hopes to offer users an uncomplicated means of handling matrices that is not too dissimilar from how they might operate more common data types. Moreover, as much of MX follows typical C and Java syntax, it hopes to provide programmers a familiar coding experience that is effortless to pick up on. Programmers will be free to decide for themselves how involved or peripheral they would like MX's matrix handling capabilities to be in their work. Lastly, MX will contain a vigorous built-in library of functions which aims to efficiently automate even the most complex matrix operations. Through implementing standard matrix operations by means of its inclusion as a data type, and providing more intricate manipulations as built-in functions, MX will supply programmers with the components necessary to construct their own complex matrix related functions.

2. Lexical conventions

a. Comments

i. The `#` character begins single-line comments

ii. `/*` begins a multi-line comment and `*/` terminates multi-line comments.

b. Identifiers

i. Identifiers are sequences of letters and digits, in which the first character has to be a letter. Inclusive of lowercase and uppercase letters. Identifiers may contain

underscores. The type of the identifier as well as its initial value must be specified upon declaration of the identifier. An identifier may be declared with a null value.

c. Key words

i. The following identifiers are reserved and may not be used otherwise

- `int`
- `bool`
- `float`
- `String`
- `continue`
- `break`
- `return`
- `Matrix`
- `if`
- `elif`
- `else`
- `new`
- `void`
- `while`
- `for`
- `true`
- `false`
- `null`
- `pi`

ii. The following identifiers are reserved for function names. A function may not be declared using the following keywords

- `main`
- `matrix`
- `numRows`
- `numCols`
- `shearV`
- `one`
- `print`
- `addRow`
- `addCol`
- `rank`
- `identity`
- `rotate`
- `dotProduct`
- `reflectY`
- `reflectX`
- `reflectO`
- `reflectYX`
- `shearH`
- `reflectNegX`

d. Strings

i. A sequence of zero or more characters (lowercase/upper case), digits, escape sequences enclosed in double quotes. Contains null character at end to indicate end of string. Strings with only one character are still considered to be variables of type String and not type char.

3. Primitive Types

a. Types

i. `int` - signed integer type, 4 bytes

ii. `float` - signed floating-point type, 8 bytes

iii. `bool` - has the value of true/false, 1 byte

iv. '`-`' will denote negative numbers (Right-associative)

b. Implicit Casting

i. If there is an operation between an *int* and *float* (arithmetic addition, subtraction, multiplication, division), then cast the *int* value to a *float*. This includes elements of a matrix.

4. Operators

a. Unary Operators

i. *`expr++`*

ii. *`expr--`*

iii. *`++expr`*

iv. *`--expr`*

b. Assignment operator

i. Values may be assigned to variables via the following syntax:

1. *`Identifier = expr`*
2. *`Type Identifier = expr`*
3. Where the value in expression will be assigned to the identifier

ii. The values of the results of arithmetic operations may also be assigned via the following syntax:

1. *`Expr += Expr`*
    a. Assignment by sum
2. *`Expr -= Expr`*
    a. Assignment by difference
3. *`Expr *= Expr`*
    a. Assignment by product
4. *`Expr /= Expr`*
    a. Assignment by quotient
5. *`Expr %= Expr`*
    a. Assignment by remainder

c. Arithmetic Operators

i. *Expr + Expr*
    1. The result is the sum of the expressions. This operation may only be performed between expressions of type *int* and type *float*. If performed between an expression of type *int* and another expression of type *float* (i.e. *int* a + *float* b), the *int* is converted to a *float* and the type of the sum is of type *float*.

ii. *Expr - Expr*
    1. The result is the difference of the expressions. The same type considerations for addition apply.

iii. *Expr * Expr*
    1. The result is the product of the expressions. The same type considerations for addition apply.

iv. *Expr / Expr*
    1. The result is the quotient of the expressions. The same type considerations for addition apply.

v. *Expr % Expr*
    1. The result is the remainder from division between both expressions. Both expressions must be of type *int*.

d. Matrix Operators

The following section details operators that are specific to MX's *Matrix* data type and the operations that can be performed between multiple expressions of type *Matrix* and, to a lesser extent, type *int* and type *float.*

i. *Matrix +. Matrix*
    1. The result is the sum of two expressions of type *Matrix*. This operation may only be performed between two expressions of type *Matrix* whose elements are of type *int* or *float*. If the operation is performed between two matrices where one is of type *int* and the other is of type *float* then the *int* matrix is cast to *float* and the resulting matrix is of type *float.*

ii. *Matrix -. Matrix*
    1. The result is the difference between two expressions of type *Matrix*. This operation may only be performed between two expressions of type *Matrix* whose elements are of type *int* or *float*. The same type considerations for matrix addition apply.

iii. *Matrix *. Matrix*
    1. The result is the product of two expressions of type *Matrix*. This operation may only be performed between two expressions of type *Matrix* whose elements are of type *int* or *float*. The same type considerations for matrix addition apply.

iv. *Matrix **. Expr*
    1. This operator indicates scalar multiplication between an expression of type *Matrix* and another expression of type *int* and type *float*. If a Matrix is populated with expressions of type *int* and multiplied by a scalar of

type *float*, the *int* expressions are converted to *float*, and the datatype of the resulting Matrix is also changed to *float*. In the converse scenario, where a Matrix is filled with expressions of *float* and multiplied by a scalar of type *int*, the *int* is converted to a *float* and the data types of the Matrix and its contained expressions remain as type *float*. Associativity is irrelevant.

v. `Matrix'`

1. This operator returns the transpose of a matrix. Return type is *Matrix*.

e. Relational Operators

The following operators are reserved for comparison between two expressions. Each operator yields 1 if the comparison is True and 0 if it is false. Comparison between expressions requires that each expression be of the same type. The last two operators have lower precedence than the first four.

i. `Expr < Expr`
ii. `Expr > Expr`
iii. `Expr <= Expr`
iv. `Expr >= Expr`
v. `Expr == Expr`
vi. `Expr != Expr`

f. Logic Operators

i. `Expr && Expr`
ii. `Expr || Expr`
iii. `!Expr`

g. Order of Precedence

Precedence	Description	Associativity
++ -- () [] .	Postfix increment and decrement Grouping or Function call Array subscripting Access matrix functions	left-to-right
- ++ -- ! ‘	Unary minus Prefix increment and decrement Logical NOT Transpose	right-to-left
* / *. **. %	Arithmetic multiplication and division Matrix-matrix multiplication, scalar-matrix multiplication Modulus	left-to-right

+ - +. -.	Arithmetic addition and subtraction Matrix addition and subtraction	left-to-right
< <= > >=	Less than, less than or equal to Greater than, greater than or equal to	left-to-right
== !=	Is equal to, is not equal to	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
= += -= *= /=	Assignment Addition, subtraction assignment Multiplication, division assignment	right-to-left
,	Separates expressions	left-to-right

5. Separators

a. Semicolon at the end of every statement (not including end of for/while loop blocks, if/else statements)
b. Curly braces in for loops, while loops, if/elif/else
c. ( ) [ ] { } ; , .
d. Ignore whitespace (don't make it a token)

6. Declarations

a. All variables should be declared with their type specification and initialized value (do not allow something like *int var1;*). A variable may be declared with a null value.

b. Declaring primitive types:

i. `type var_name = value;`

c. Declaring Matrix objects

1. There are two ways to declare a Matrix object:
   a. `Matrix m = datatype [[r1], [r2], [r3], … [rn]]`
   Here, we create a matrix with values where `r1, r2,...` represent rows (1D arrays) of the matrix. The elements of r1, r2.. should be of the same type

   b. `Matrix n = datatype matrix(int m, int n)`
   This creates an empty matrix with the dimensions `numRows` by `numCols`. Its elements are of type *int* or *float*. Sets elements to default values (0 for a matrix of integers, 0.0 for a matrix of floats).

d. **Functions**

i. All functions must be defined when being declared. Function names include letters/digits (lowercase/uppercase) and when declaring functions, it should specify the return type as shown below in *datatype*:

```
datatype foo(datatype parameter1, … , datatype
parameter_n) {

}
```

All functions are public.

ii. Every MX program should contain a `main()` function, which starts every program.

7. Statements

a. Expression statements

i. Expression statements take the form of "*expr;*"

b. Conditional statements

i. Conditional statements may take the forms of:

```
1. If ( expr ) { stmt; }
2. If ( expr ) { stmt; } else { stmt; }
3. If ( expr ) { stmt; } elif ( expr ) { stmt; }
4. If ( expr ) { stmt; } elif ( expr ) { stmt; }
   else { stmt; }
```

c. While statement

```
i.  while ( expr ) {
        stmt;
    }
```

d. For statement

```
i.  for( expr_1, expr_2, expr_3 ) {
        stmt;
    }
```

e. Return statement

i. If a function is of type void (As declared in its declaration), a return statement is not required. Otherwise, a return statement is required.

```
return expr;
```

f. Break statement

i. Used to terminate *while* and *for* loops. May only be written inside of a *while* or *for* loop

```
break;
```

g. Continue

i. Force starts the next iteration of a *while* or *for* loop. May also only be written inside of either mentioned loop.

```
continue;
```

h. Null statement

i. Data Type includes `String, int, float` and `matrix`.

```
datatype var_name = NULL;
```

8. Scope

Declarations made within functions are visible only within those functions (i.e. their scope). A declaration is not visible to declarations that came before it. You cannot declare an already declared variable, but redefining variables is allowed.

9. Function calls

a. Functions may be called using the following syntax:
`function_name(parameter_1, … , parameter_n )`

10. Sample Code

3.1 Basic syntax: example of a user defined function for determining the greatest common divisor of two integers

```
int gcd(int x, int y)
{
      # example of a simple user-defined function
      while (x != y)
      {
            if (x > y)
                  x -= y;
            else
                  y -= x;
      }
      return x;
}

int main ()
{
    int x = 3;
    int y = 15;
    int z = gcd(x, y);
    printf("%d", z); # prints 3
    return 0;
```

```
}
```

3.2 Simple program illustrating built in declaration and manipulation of matrices in our language

```
int main()
{
      Matrix m1 = [[0, 1], [2, 3]; # matrix declaration
      m1.print();
/* prints the following
[0, 1]
[2, 3]
*/

      Matrix m2 = [[3, 4], [4, 5]]; # matrix declaration
      m2.print();
/* prints the following
[3, 4]
[4, 5]
*/

      Matrix m3 = m1 *. m2;
      m3.print();
/* prints the following
[4, 5]
[1, 2]
[8, 3]
*/

      Matrix m4 = m1' +. m2;
      m4.print();
/* prints the following
[3, 6]
[5, 8]
*/

      return 0;
}
```

3.3 C-program approximation of matrix manipulation. As you can see, our language will improve the way in which matrices are added, subtracted, etc. (less lines of code).

```
#include <stdio.h>
#include <stdlib.h>
```

```
void add(int m[2][2], int n[2][2], int sum[2][2])
{
    for(int i = 0; i < 2; i++)
        for(int j = 0; j < 2; j++)
            sum[i][j] = m[i][j] + n[i][j];
}

void multiply(int m[2][2], int n[2][2], int res[2][2])
{
    for(int i = 0; i < 2; i++)
    {
        for(int j = 0; j < 2; j++)
        {
            res[i][j] = 0;
            for (int k = 0; k < 2; k++)
                res[i][j] += m[i][k] * n[k][j];
        }
    }
}

void transpose(int matrix[2][2], int trans[2][2])
{
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            trans[i][j] = matrix[j][i];
}

void print_matrix(int matrix[2][2])
{
    for(int i = 0; i < 2; i++)
    {
        printf("[");
        for(int j = 0; j < 2; j++)
        {
            printf("%d", matrix[i][j]);
            if(j < 1)
                printf("\t");
        }
        printf("]\n");
    }
}

int main()
{
```

```
    int m1[2][2] = {{0, 1},{2, 3}};
    int m2[2][2] = {{3, 4},{4, 5}};
    int m3[2][2];
    print_matrix(m1);
    printf("\n");
    print_matrix(m2);
    printf("\n");
    multiply(m1, m2, m3);
    print_matrix(m3);
    printf("\n");
    transpose(m1, m3);
    add(m3, m2, m3);
    print_matrix(m3);
    printf("\n");

    return 0;
}
```