# MX

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. Language Syntax**

**2.1 Scope**

{} for functions, blocks, classes

; for end of line

**2.2 Comments**

```
 # for single line comments
/* for multi line comments */
```

**2.3 Conditional**

```
if(...) {

} elif(...) {

} else {

}
```

**2.4 Loops**

```
for( int i = 0; i < 5; i++) {
      while(conditional){
            break;
      }
      continue;
}
```

**2.5 Variable Declaration**

```
int i = 3;
double j = 4.0;
char star = ‘a’;
String string = “shark”;
```

There are two ways to declare a Matrix object:

```
Matrix m = [r1, r2, r3, ...]
```

Here, we create a matrix with values where `r1, r2,...` represent rows (1D arrays) of the matrix.

```
Matrix n = datatype matrix(int m, int n)
```

This creates an empty matrix with the dimensions `numRows` by `numCols`. Its elements are of type `datatype` (int, double, or float). Sets elements to default values (0 for a matrix of integers, etc.).

Note: Matrices will be stored on the heap.

```
N[row][column] # gives you an element in the matrix
```

**2.6 Data Types**

```
int k;
double g;
float f;
char a;
String plt;
Matrix m;
```

**2.7 Arithmetic Operators**

| Operators | Description | Examples |
|---|---|---|
| + | Arithmetic Addition | `1 + 3 # literals`<br>`x + 8 # var and lit`<br>`x + z # two var sum` |

| | | |
|---|---|---|
| `-` | Arithmetic Subtraction | `1 - 3 # literals`<br>`x - 8 # var and lit`<br>`x - z # two var sub` |
| `/` | Arithmetic Division | `1 / 3 # literals`<br>`x / 8 # var and lit`<br>`x / z # two var div` |
| `*` | Arithmetic Multiplication | `1 * 3 # literals`<br>`x * 8 # var and lit`<br>`x * z # two var multi` |
| `%` | Modular Arithmetic | `1 % 3 # literals`<br>`x % 8 # var and lit`<br>`x % z # two var sum` |
| `++x` | Preincrement Operator | `int x = 0;`<br>`++x;` |
| `x++` | Postincrement Operator | `int y = 1;`<br>`y++;` |

The `+`, `-`, and `*` operators can also be used for addition, subtraction, and multiplication of matrices. The * operator will be used for matrix-matrix multiplication and scalar-matrix multiplication.

### 2.8 Relational Operators

| Operator | Description | Examples |
|---|---|---|
| `==` | Equal to | `int x = 3;`<br>`if (x == 3) {}` |
| `!=` | Not Equal | `int y = 3;`<br>`if(y != 4) {}` |
| `>` | Greater Than | `if(4 > 5) {}` |
| `<` | Less Than | `while(5 < 14) {}` |
| `>=` | Greater Than Or Equal To | `while(i >= 0) {}` |
| `<=` | Less Than Or Equal To | `while( f <= 10) {}` |

### 2.9 Logic Operators

| Operator | Description | Example |
|---|---|---|
| `&&` | Logical and | `while(x>3 && y>4){}` |
| `\|\|` | Logical or | `while(x>3 \|\| y>4){}` |

| | | |
|---|---|---|
| `!` | Logical negation | `while(x>3 && y!=4){}` |

**2.10 Functions**

Declaration:

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

}
```

Call:

```
foo(parameter1, ...)
```

**2.11 Built-In Functions**

| Function | Return type | Description |
|---|---|---|
| matrix(int m, int n) | Matrix | Returns an empty mxn matrix |
| numRows() | int | Returns the number of rows in a matrix |
| numCols() | int | Returns the number of columns in a matrix |
| zeros(int n) | Matrix | Returns an nxn matrix filled with zeros of type integer |
| ones(int n) | Matrix | Returns an nxn matrix filled with ones of type integer |
| print(parameter) | void | Prints the value passed in *Prints a Matrix parameter row by row |
| addRow(int index, datatype[] arr) | bool | Adds a row specified by datatype[] arr (1D array) to the matrix at the row index *index.* Returns true if possible, else false. |
| addCol(int index, datatype[] arr) | bool | Adds a col specified by datatype[] arr (1D array) to the matrix at the column index *index.* Returns true if possible, else false. |
| rank() | int | Returns the rank of a matrix |
| identity(int n) | Matrix | Returns an nxn identity matrix |

| | | |
|---|---|---|
| rref() | Matrix | Returns the rref of a matrix |
| transpose() | Matrix | Returns the transpose of a matrix |
| dotProduct(Matrix n) | int, double, or float | Returns the dot product of a matrix and matrix n |
| rotate(double angle) | Matrix | Returns the rotation of a matrix about an angle |
| reflectX() | Matrix | Returns the reflection of a matrix over the x-axis |
| reflectY() | Matrix | Returns the reflection of a matrix over the y-axis |
| reflectYX() | Matrix | Returns the reflection of a matrix over the line y=x |
| reflectO() | Matrix | Returns the reflection of a matrix about the origin |
| reflectNegX() | Matrix | Returns the reflection of a matrix over the line y=-x |
| shearH(int k) | Matrix | Returns the horizontal shear of a matrix by a factor of k |
| shearV(int k) | Matrix | Returns the vertical shear of a matrix by a factor of k |

Note: Many of these functions are called as follows:

```
m.addRow(3, [3, 2, 1]) # adds a row at index 3 of Matrix m
```

These functions include: numRows(), numCols(), addRow(), addCol(), rank(), rref(), transpose(), dotProduct(), rotate(), reflectX(), reflectY(), reflectYX(), reflectO(), reflectNegX(), shearH(), shearV().

Before these operations are carried out, the compiler will first check that they can be done on those matrices given their dimensions. Throws an error otherwise.

**2.12 Reserved Words**

break, continue, bool, int, double, float, char, String, Matrix, if, elif, else, new, return, void, while, for, true, false, null, return

## 3. Sample Algorithms

**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
```

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

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

$$\begin{bmatrix} 3 & 4 \\ 4 & 5 \end{bmatrix}$$

```
      Matrix m3 = m1 * m2;
      m3.print();
# prints the following
```

```
[ 4   5 ]
[ 1   2 ]
[ 8   3 ]

     Matrix m4 = m1.transpose() + m2;
     m4.print();
# prints the following
[ 3   6 ]
[ 5   8 ]

     return 0;
}
```

**3.3** C-program approximation of matrix manipulation

```
#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];
}
```

```c
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;
}
```