J-STEM: Matrix Manipulation Language

Language Reference Manual

Tessa Hurr (trh2124)
Julia Troxell (jst2152)
Emily Song (eks2138)
Samantha Stultz (sks2200)
Michelle Lu (ml3720)

Table of Contents

1. Introduction

J-STEM is a 2-D matrix manipulation language. Matrices are important tools in mathematics with various applications such as graphic rendering and encryption. We have found that matrix manipulation in Java is tedious and difficult. We propose J-STEM, a language that allows for easy and intuitive matrix transformations. J-STEM is compiled into LLVM.

The matrix manipulation language will have functions to transform matrices. The functions will include calculating the determinant, rotating the matrix, applying operations to each value in the matrix, and deleting or adding columns and rows. Basic functions to add, subtract, multiply, and divide matrices will be implemented as well. The goal of our language is to utilize these functions and apply them to images. The language will be able to edit various features of images. These include different color overlays, image rotations, and altering contrast and brightness.

2. Types

Data Type	Description	Declaration
mx	A matrix (a list of rows), with type specified Default initialization to a 0x0 matrix (an empty list)	`int mx m = matrix();` `float mx m = matrix(1, 2);`
row	A list	`row r = [1, 2, 3];`
string	String value	`string str = "hello!";`
int	Integer value	`int x = 5;`
float	Floating point value	`float x = 5.0;`
column	A list	`col c = [1, 2, 3];`
file	file	`file filename = file.ppm;`
list	Like Java	`list l[];`
pixel	A tuple of length 3; a wrapper for a tuple	`pixel p = (255, 255, 255);`

tuple	Like Java	`tuple t = (1, 2, 3);`
bool	value True or False	`bool b = True;`

3. Lexical Conventions

3.1 Identifiers

An identifier must begin with a lowercase or uppercase character, and can otherwise consist of any combination of these characters along with digits and underscores. Other symbols (i.e. !, @, #, etc.) are prohibited in variable names, and keywords (i.e. for, while, if, etc.) are prohibited as variable names as well.

3.2 Keywords

Keyword	Description	Usage
print()	Prints to console	`print(m);`
for	Iteration until condition is reached	`for(condition) { … }`
while	Loop until condition is reached	`while(condition) { … }`
range	Range of numbers used in a for loop	`range(0, 9, 2)` `// 0, 2, 4, 6, 8`
if	If in an if-else statement	`if { … }`
else	Else in an if-else statement	`else { … }`
else if	Additional if statement after an if-else statement	`else if { … }`
return	Returns value from a function	`return m;`
main	Main function	`def void main() { … }`
def	function declaration	`def void x(int y) { … }`
in	Traverse through a sequence in an enhanced matrix for loop	`for(row in matrix){` `  ...process row...`

		`}`
void	Return type for a function that does not return anything	`def void no_return() { … }`

3.3 Comments

#	single-line comment
/* … */	multi-line comment

3.4 Operators

Arithmetic Operators	**Description**	**Usage**
=	Assignment operator	`int y = 6, int z = 2`
+	Arithmetic operators	`int x = y + z` *// x = 8*
-	Subtraction operator	`int x = y - z` *// x = 4*
*	Multiplication operator	`int x = y * z` *// x = 12*
/	Division operator	`int x = y / z` *// x = 3*
^	Exponentiation operator	`int x = 2^3` *// x = 8*
==	Returns 1 if the values are equal, 0 otherwise	`y == z` *// return 0*
+=	Adds the value on the left to the value on right and stores in left variable	`y += 1` *// y = 7*
!=	Returns 1 if the values are not equal, 0 otherwise	`y != z` *// return 1*
>	Greater than operator	`y > z` *// return 1*
<	Less than operator	`y < z` *// return 0*
>=	Greater than or equal to operator	`y >= z` *// return 1*
<=	Less than or equal to operator	`y <= z` *// return 0*
&&	Logical AND operator	`0 && 1` *// return 0*

||	Logical OR operator	`0 && 1` *// return 1*
!	Logical NOT operator	`!(5 == 5)` *// return 0*

Matrix Operator	**Description**	**Usage**
++, --, **, //, =?	Scalar and matrix operations	`M = 5 * M1` *// scalar multiplication* `M = M1 ** M2` *// matrix multiplication*
==, !=, >, <, >=, <=	If two matrices have the same dimension: Compare element in one matrix to corresponding element in other matrix, for entire matrix	`M1 == M2` *// returns 1 if matrices contain same elements, 1 otherwise*
M[r1:r2][:]	Access rows from r1 (inclusive) to r2 (exclusive) of matrix M. This is similar to list slicing in Python.	`M[2][:]` *// access row 2* `M[2:4][:]` *// access rows 2-3* `M[2:][:]` *// access rows 2-len*
M[:][c1:c2]	Access columns from c1 (inclusive) to c2 (exclusive) of matrix M. This is similar to list slicing in Python.	`M[:][4]` *// access column 4* `M[:][4:8]` *// access column 4-7* `M[:][4:]` *// access column 4-len*

3.5 Precedence

Precedence	**Operators**
Highest	Function and matrix declarations
	!
	*, /
	+, -
	<, > , <=, >=
	==, !=
	&&

	||
Lowest	=

4. Standard Library Functions

Function	Description	Usage

addRow(int rowNum, row r)	Insert a row of 0's at the bottom of the matrix (end of list)
	If optional arg1 is supplied, insert a row of 0's at that index
	If arg1 and arg2 are supplied, insert arg2 (a row) and index arg1
		`m.addRow();`
		`m.addRow(2);`
		`m.addRow(2, [1,2,3]);`

delRow(int rowNum)	Delete last row in matrix (end of list) If optional argument supplied, delete row at that index	`m.delRow();` `m.delRow(2);`
matrix(int x, int y)	Create a matrix Arguments specify #rows, #columns No arguments creates a 0x0 matrix	`int mx m = matrix();` `int mx m = matrix(2, 3);`
loadFile(file filename)	Load an image file into a matrix	`int mx m = loadFile(file.ppm);`
transpose()	Transpose matrix	`m.transpose();`
inverse()	Inverse matrix	`m.inverse();`

addColumn(int colNum, row r)	Insert a column at the right of the matrix
	If optional arg1 is supplied, insert a column of 0's at that index
	If arg1 and arg2 are supplied, insert arg2 (a column) and index arg1
		`m.addColumn();`
		`m.addColumn(3);`
		`m.addColumn([1,2,3]);`

delColumn(int colNum)	Delete rightmost column in matrix	`m.delColumn();` `m.delColumn(3);`

	If optional argument is supplied, delete column at that index	
length()	Returns # of rows in matrix or # of elements in a row	`int m_len = m.length();` `int row_len = m[0].length();`
append()	Adds another row to the end of a matrix	`m.append([2, 3]);`

5. Syntax

5.1 Expressions

Arithmetic and Matrix Operations

Assignment operators are binary operators with right-to-left associativity. Arithmetic and matrix expressions are mathematical operations with left-to-right associativity.

Scalar multiplication

$M = 5 * M_1$

Matrix multiplication

$M = M_1 ** M_2$

5.2 Declaration and Initialization

Variable Declaration and Initialization

All variables must be declared before use. A declaration specifies the variable type and the variable name. A variable may also be initialized in its declaration. A variable can also be declared in one line and initialized in the next.

General Examples

```
variable_type variable_name;

variable_type variable_name = literal;
```

```
variable_type variable_name;
variable_name = literal;
```

Specific Examples

```
int x;

int x = 6;

int x;
x = 6;
```

Matrix Declaration and Initialization

All elements in a matrix must be of the same type, and elements in a matrix can only be ints and floats. A proper declaration specifies the element type, that the variable is a matrix, and the matrix name. “matrix()” initializes an empty matrix, “matrix(r, c)” initializes a matrix with r rows and c columns, and functions such as addRow and addColumn can be used to populate/edit the matrix.

General Examples

```
element_type mx matrix_name;

element_type mx matrix_name = matrix();

element_type mx matrix_name;
matrix_name = matrix();
```

Specific Examples

```
int mx M;

int mx M = matrix();

int mx M;
M = matrix()
```

A matrix can also be initialized with the following syntax:

```
int mx M = {(1, 2, 3), (4, 5, 6)};
```

6. Control Flow

6.1 Statements and Blocks

Each statement is followed by a semicolon.

Example:

```
int x = 6;
```

Blocks are surrounded by brackets.

Example:

```
if (x == 6) {
    print("x is 6");
} else {
    print("x is not 6");
}
```

6.2 If-Else If-Else

if-else if-else statements are blocks. When the "if" condition is not met, the program will check for any other conditions (specified by "else if" or "else"). "else if" is required if checking for more than two conditions; "else" will suffice otherwise. When a condition is met, the program will execute the code in the corresponding block, and ignore all subsequent "else if" and "else" conditions. An "else if" statement requires a condition, while an "else" statement does not.

Example:

```
if (x > 6) {
    print("x is greater than 6");
} else if (x < 6) {
    print("x is less than 6");
} else {
    print("x is equal to 6");
}
```

6.3 Loops

For Loop

The J-STEM for loop operates like Java's. There are 3 fields to the condition of a for loop. First, an index variable is initialized to some value, then the stop condition is specified, and then the increment/decrement of the index variable is given (using operators += or -=). The block is looped through according to these fields.

```
for ( index, stop_condition, step_value) {
     ...
}
```

Matrix For Loop

An enhanced version of the standard for loop used to iterate easily through matrices. The enhanced for loop uses the keyword "in" to iterate through a list type (i.e. a matrix, a row, etc.)

```
for ( elt1 in iterable ) {
     …
}
```

While Loop

The while loop also operates like Java's. In the condition, there is only one field, for specifying the stop condition. Usually, this stop condition utilizes operators such as ==, <, >, etc.

```
while ( stop_condition ) {
     ...
}
```

Examples:

```
for (int i = 0; i < 10; i += 1) {
     ...
}

for (row in matrix) {
     for (cell in row) {
          …process cell...
     }
}

while (i < 10) {
     ...
```

```
}
```

6.4 Break and Continue

Break Statement
The break statement terminates a loop.

```
while ( stop_condition ) {

     break; # program exits loop

}
```

Continue Statement
The continue statement skips the rest of the current iteration of the loop, and starts the next iteration of the loop.

```
while ( stop_condition ) {

     continue; # program jumps to next iteration of the loop

}
```

7. Functions and Program Structure

7.1 Functions and Function Calls

To declare functions, use keyword ‘def’.If there is no return value for the function, use keyword ‘void’ and ‘main’ followed by parenthesis and brackets for the function body. If the function returns a value, state the type of the return of the function, function name, and argument type and names in parenthesis followed by brackets to enclose the function body. Functions have to be defined as such and implemented before being called.

Function Declaration

```
def void main ( ) {
     statement;
}
```

```
def  return_type function_name  ( arg_type arg_name ) {
     return expression;
}
```

Example:

```
def mx add (mx matrix_a, mx matrix b) {
     new_matrix = a ++ b;
     return new_matrix;
}
```

7.2 Scope Rules

The scope of a variable depends on when it is declared within a function. If the variable is declared at the outermost level of the function (i.e. at the beginning of the function), then it is accessible throughout the function. If the variable is declared at the beginning of a loop, then it is only accessible within that loop. Likewise, if the variable is declared as a variable in the condition of a loop, then it is also only accessible within that loop.

Example 1

```
def void main() {
    int x = 5;
    print(x); # will print 5
}
```

Example 2

```
def void main() {
    int i = 0;
    while(i < 5) {
     int square = i * i;
     print(square); # will print 0, 1, 4, 9, 16
     i = i + 1;
    }
    print(square); # error
}
```

Example 3

```
def void main() {
```

```
    for(int i = 0; i < 5; i++) {
     print(i)  # will print 0, 1, 2, 3, 4
    }
    print(i); # error
}
```

7.3 Block Structure

Functions, conditionals, and loops will all be written using block structure, defined by an opening bracket "{" and a closing bracket "}". Variables declared within brackets cannot be accessible outside the block.

Example Code

This is example code in our language that multiplies two matrices. We have a matrix multiplication operator (**) in our language, but this manual implementation is a good way to see how some of the parts of our language work.

```
/* multiply 2 matrices */
def int mult(row1, col2) {
     int sum = 0;
     for(int i = 0; i < row1.length(); i += 1){
          sum += (row1[i] * col2[i]);
     }
     return sum;
}

def void main() {
     int mx m1 = {(1,2), (3,4)};
     int mx m2 = {(2,3), (4,5)};
     int mx m3 = matrix(m1.length(), m2[0][:].length());
     for(row_m1 in m1) {
          row m3_row = [];
          for(col_m2 in m2) {
               m3_row.append(mult(row_m1, col_m2));
          }
          m3.addRow(m3_row);
     }
     print(m3);
}
```