MatriCs: The Ultimate Matrix Manipulation Language
Language Reference Manual

Short name: MaC
Extension: .neo

Talal Asem Toukan [tat2132] - Manager
Emmanuel Koumandakis [ek2808] - System Architect
Duru Kahyaoğlu [dk2565] - Language Guru
Florian Shabanaj [fs2564] - Language Guru
Nikhil Raghav Baradwaj [nrb2129] - Tester

Table of Contents

1 Introduction

Matrices are crucial tools in representing and manipulating finite sets of data across a wide range of subject matter, including the life sciences, mathematics, engineering, and computer science. To add to that, basic matrix operations are unreasonably difficult to perform in existing languages without the use of external packages in languages like Python and R. Therefore, the purpose of our language, MatriCs, is to streamline that process by simplifying those computations, while simultaneously reducing the running time of basic operations. Using syntax like that of common high-level languages, MatriCs is tailored to programmers familiar with those languages, but not necessarily familiar with data science and matrix packages like SciPy (Python), NumPy (Python), and Matrix (R).

MatriCs main advantage is **automatic parallelization**. This special feature ensures that we preserve the hardware/software division paradigm while taking advantage of processors with SIMD (single instruction multiple data)/Vectorization capabilities. Since MatriCs compiles to an LLVM intermediate representation, we can take advantage of LLVM's automatic parallelization infrastructure. The programmer can then write sequential instructions and have the compiler generate vector instructions. This is, however, only done for the built-in matrix functions and operators but it is a significant improvement for most common linear algebra applications.

MatriCs is a strongly typed language that combines a high-level syntax with a whole host of special operators. These operators enable the user to perform fundamental computations involving linar algebra, including but not limited to calculating a matrix's transpose and inverse, carrying out matrix multiplication, and extracting sub-matrices through slicing. At the very core of our language is the special data type: the matrix. MatriCs will compile to LLVM.

2 Lexical Elements

2.1: Tokens

Our language can be broken down into six categories of tokens: identifiers, keywords, literals, operators, punctuation, and comments. Whitespace is used to separate tokens but otherwise will be ignored. Indentation should be used for stylistic purposes but is not necessary for the proper functioning of MatriCs programs.

2.2: Identifiers

Identifiers are strings used for naming different elements, such as variables and functions. Identifiers must begin with a letter, but can contain digits and underscores as well.

These rules are described by the definitions involving regular expressions below:
identifier := (letter) (letter | digit | underscore)*
digit := '0' - '9'
letter := uppercase_letter | lowercase_letter

uppercase_letter := ‘A’ - ‘Z’
lowercase_letter := ‘a’ - ‘z’

2.3: Keywords

The following literals cannot be used as identifiers. They are also case sensitive.

Syntax	*Description*
if	Similar to C conditional
elif	Similar to C conditional (but with `elif` keyword instead of `else if`)
else	Similar to C conditional
for	Similar to C for loop
foreach	Enhanced for loop that executes action for each element in the matrix
while	Similar to C while loop
return	Return from function
null	No data
void	Returns nothing
import	Import external libraries for extended functions

2.4: Punctuation

Symbol	*Purpose*
;	Used to end a statement, as well as to define each row of a matrix.
{ }	Curly brackets are used to enclose functions, while and for loops, and if statements. In other words, they are used to delineate the scope of blocks of code in the program. They are also used to define the special data type, matrix.
( )	Use to specify and pass arguments for a function and the precedence of operators. Also used to enclose conditions in for and while loops and if statements.
[ ]	Use to specify dimension of the matrix data-type when it is declared. Also used to access elements in the array at a given position. ‘

,	Used to separate function arguments and to separate different in a specific row in a given matrix.
“ ”	Used to declare a variable of string data type
//	Inline comment
/* */	Block comment

2.5: Operators and Expressions

2.5.1: Standard Operators

Name	*Syntax*	*Example*
Addition,Subtraction, Multiplication, Division, Modulo	+,-,*,/,%	int a = 5 + 8 //a = 13

Additive, Subtractive, Multiplicative, Divisive, Modular Assignment	+=, -=, *=, /=, %=	int a = 4;
		a += 2;
		//a = 6

Assignment	=	int a = 7 //a has a value of 7
Equality check	==	7 == 7 //Returns 1
Greater than	>	6 > 5 //Returns 1
Less than	<	5 < 3 //Returns 0
Greater than or equal to	=>	5 => 4 //Returns 1
Less than or equal to	<=	5 <= 5 //Returns 1
Not equal	!=	5 != 3 // Returns 1

Logical Not	!	int a = 1;
		int b = 0;
		!(a && b) //Returns 1

Logical AND	&&	//with the values from the example above a && b //Returns 0
Logical OR	||	//with the values from the example above a || b //Returns 1

2.5.2 Matrix Operators

Name	*Description*	*Syntax*	*Example*

Scalar Multiplication, Scalar Division, Scalar Power	Element-wise multiplication/ division or scalar multiplication/ division/power	.*, ./, .^	mat int C = A.*B;
			mat int C = A./B;
			mat int C = A.*2;
			mat int C = A./2;
			mat int C = A.^2;

Matrix Multiplication, Matrix Division	Matrix multiplication/ division.Operation is not commutative. If at least one input is scalar, then A*B is equivalent to A.*B and is commutative.	*, /	mat int C = A*B;
			mat int C = A/B;
			mat int C = A*3;
			mat int C = A/3;

Addition, Subtraction	Addition/ subtraction, scalar or element-wise	+, -	mat int C = A+B;
			mat int C = A+2;
			mat int C = A+B;
			mat int C = A-2;

Transpose	Returns the transpose of a matrix	’	mat int B = A’;
Indexing	Returns the element in the specified row and column of a given matrix	matr_x[*row_index*][*column_index*]	matr_x[2][4] //returns the element in the 3rd row and 5th column of the given matrix
Slicing	Returns an array of elements with the	matr_x[row_index1: row_index2,	matr_x[1:2,2:4] //returns the matrix

	specified location in terms of rows and columns	column_index1:column_index2]	in the 2nd to 3rd rows and 3rd to 5th columns

2.5.3: Expressions

Expressions are made of at least one operand and zero or more operators. Innermost expressions are evaluated first and the priority of an expression is determined by parentheses. The direction of evaluation is from left to right.

2.6: Literals

2.6.1: String Literals

String literals are a sequence of zero or more letters, spaces, digits, other ASCII characters numbers 32 to 126, excluding the double quote. These strings should be enclosed in double quotes. For example: “Hello, world”.

2.6.2: Integer Literals

Integer literals are one or more number digits, in succession with no whitespace or punctuation character in between them. For example: 42, 666, 2.

2.6.3: Floating-Point Literals

Floating-point literals are decimal numbers. A decimal number is a fraction whose denominator is a power of ten and whose numerator is expressed by figures placed to the right of a decimal point. The integer part is expressed by figures placed to the left of a decimal point. Similarly to integer literals whitespace is not allowed to separate digits or digits and the decimal point `.`. For example: 4.2, 5.0, 222.666

2.6.4: Matrix Literals

Matrix literals can be integer or floating-point numbers. Matrices can only be composed of entirely integers or floating-point numbers. For example: [1,2,3; 4,5,6; 7,8,9; 10,11,12], [1.0,1.5; 2.0,2.5; 3.0,3.5]

3 Data Types

3.1: Primitive Data Types

MatriCs provides primitive data types that are common to many high level languages. The full list of primitive data types is given below:

Type	*Description*	*Syntax*	*Range of Values*	*Example*
integer	Used to define integer types	int	-2,147,483,648 to 2,147,483,648	int a = 5;
float	32-bit floating number	float	±1.7976931348 6231570E+308	float a = 5.0;
bool	Used to define boolean types (true and false)	bool	True, False	bool flag = true;
string	Used to define strings of characters	string	Any string of ASCII characters enclosed, excluding double quotes.	string str = “abcd1234”;

3.2: Non-Primitive Data Types

3.2.1: Matrices

Note that MatriCs doesn’t have a data type called arrays and therefore the matrix data type can be used to declare/define an array of any dimensions (i.e. the matrix data type can be used to declare/define one dimensional array).

Type	*Description*	*Syntax*	*Example*
matrix	Defines a matrix	mat	mat int matr_x = {1,2,3; 4,5,6; 7,8,9};

3.2.2: Declaring Matrices

A matrix is declared by first typing “mat” followed by the data type to be stored in the matrix, the name of the matrix, and the dimensions of the matrix enclosed by square

brackets. An example is given below:

mat int matr_x [2][5]; //returns a 2 by 5 uninitialized matrix holding values of type int
mat int matr_x = {0, 0, 0; 0, 0, 0}; //declares a 2 by 3 and initializes its values to zero

3.3.3: Accessing and Setting Array Elements

A matrix element can be accessed by simply typing the name of the matrix and the dimensions of the element desired to be accessed inside square brackets.

int elmt = matr_x[2][5];
//elmt is equal to 6 - i.e. the element in the second row and fifth column of "matr_x"

Array elements can be set either in the declaration stage or later on. Simultaneous declaration and definition is demonstrated in the example below:

mat int matr_x = {1,2,3; 4,5,6; 7,8,9};

Array elements can also be set after declaration. An example is given below:

matr_x[2][5] = 99

4 Statements

4.1: The if Statement

Format:

```
if(condition){
        action1;
}elif(condition){
        action2;
}else{
        action3
}
```

Description:
The if statement consists of a block of code that only executes if the condition enclosed within the parentheses is true. If not, the block of code is ignored and the program jumps to the next line. Optional additions include elif, which contains another condition to be checked, and else, which executes in the case that none of the conditions above are met. These additions are not required, but if an else statement is used, then it should be the last element of the statement.

4.2: The while Loop

Format:

```
while(condition){
        action;
}
```

Description:

The while loop consists of a block of code that repeatedly executes as long as the condition enclosed within the parentheses is true. If the condition is not true when the code is first encountered, then the program jumps over the block entirely. Furthermore, if the condition is true and remains true indefinitely, then the code gets caught in an infinite loop and the program never continues beyond the block.

4.3: The for Loop

Format:

```
for(variable initialization; condition; increment step){
        action;
}
```

Description:

The for loop is a generalization of the while loop. Within the parentheses there are three distinct parts, separated by semicolons. The first part, variable initialization, runs once when the for loop is first encountered. The second part is a condition checked on every iteration to determine whether the block of code inside the loop should be executed. If it is executed, the third part then increments (or decrements) and the condition is re-evaluated. The initialization, condition, and increment can be any expressions.

4.4: The Enhanced foreach Loop

Format:

```
foreach(element e : matrix){
        action;
}
```

Description:

The foreach is an enhanced version of the for loop that serves the specific purpose of iterating over a matrix and performing an action for/on every element of that matrix. Within the parentheses there are two parts: the matrix to be iterated over, and each element e that can be accessed within the block of code. The syntax here is less flexible

than the ordinary for loop. Only foreach loops resembling the one above will compile.

5 Functions

5.1 Function Definitions

Function definition consists of the type of value returned by the function followed by the name of the function and the set of parameters and types enclosed in parentheses. The scope of the function will be defined by the opening and the closing curly braces, one placed after the function declaration and the other placed after the block of code defining the function is finished. An example of a function is given below:

```
int determinant(mat int m){
    int det = 0;
    int sign = 1;

    if(rows(m) == 1) {    // base case, dimensional array
      return m[0][0];
    }
```

5.2 Calling Functions

A function can be called by its name followed by its parameters. Note that there is no need to specify the type of the parameter when calling the function. An example is given below:

sum (2, 4) //returns the sum of 2 and 4

6 Program Structure and Scope

6.1 Program Structure

MatriCs program are found on a single source file. The major components of a MatriCs program are matrix declarations, matrix specifications, and function declarations in this specific order.

6.2 Scope

Declarations made within a while/for loop, if statement, or any function are available only by reference within this specific block of code. Ones that are made outside of while/for loops, if statement, or any function are available by reference throughout the rest of the code.

7 Built-in Functions

7.1 The print Function

`void print(string str)`: prints string str on the console

7.2 The dimension Function

```
mat int dim(mat int matrix)
```
OR
`mat int dim(mat float matrix)` : returns a one dimensional matrix (array) consisting of two integers {number of rows; number of columns}

7.3 The identity Function

`mat int identity(int size)` : returns a square `size` by `size` identity matrix

7.4 The zeroes Function

`mat int identity(int rows, int cols)`: returns a `rows` by `cols` matrix with all values initialized to 0

8 Standard Library

The standard library is not a part of MatriCs, but an environment that supports standard MatriCs will provide the function declarations and type and macro definitions of this library. The standard library is invoked by calling "`import stdlib`". Currently, the MatriCs standard library contains the following functions:

```
print("string input", const char *format)
```

`*format` is an optional character to indicate the kind of data being printed, especially when the print statement involves at least two kinds of data types. This is similar to C or Java, where format of an integer being printed might be indicated by %d.

Appendix A Sample Code

Methods may be used recursively:

```
int determinant(mat int m){
    int det = 0;
    int sign = 1;

    if(rows(m) == 1) {    // base case, dimensional array
      return m[0][0];
    }

    // lib functions to get # of rows and columns in m
    // (actually the values should be the same anyway)
    int rows = rows(m);
    int cols = cols(m);

    // finds determinant using row-by-row expansion
    for(int i = 0; i < rows; i++){

      // keep decomposing the matrix by 1 dimension
      mat int smaller_m[rows-1][cols-1];

      for(int a = 1; a < rows; a++){
        for(int b = 0; b < cols; b++){

          if(b < i){
            smaller_m[a-1][b] = m[a][b];
          }
          elif(b > i){
            smaller_m[a-1][b-1] = m[a][b];
          }
        }
      }

      if (i%2 == 0){    // sign changes based on i
        sign = 1;
      }
      else{
        sign = -1;
      }
      det += sign*m[0][i]*determinant(smaller_m); // rec call
for det
```

```
    }
    return det;
}


mat int transpose(mat int m) {
    int rows = rows(m);
    int cols = cols(m);

    mat int m_transpose[cols][rows];

    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < cols; j++) {
            m_transpose[j][i] = m[i][j];
        }
    }

    return m_transpose;
}


int sum_matrix(mat int m) {
     int sum = 0;
     foreach(int e : m) {     //Will go through all elements
          sum += e;
     }
     return sum;
}
```

Appendix B Context-Free Grammar

program → **ε** | program vdecl | program fdecl

fdecl → **id** (formals) { vdecls stmts }

formals → **id** |formals , **id**

vdecls → vdecl | vdecls vdecl

vdecl → type **id**; | type **id** = expr; | matrix **id** | matrix **id** = matexpr;

stmts → **ε** | stmts stmt

stmt → expr ; | return expr ; | { stmts } | if (expr) stmt else_if | if (expr) stmt else_if
else stmt | while (expr) stmt | for (expr ; expr ; expr) stmt | foreach(vdecl : id)

expr → **lit** | **id** | **id** (actuals) | (expr) | expr + expr | expr - expr | expr * expr | expr / expr
| expr == expr | expr != expr | expr < expr | expr <= expr | expr > expr | expr >=
expr | expr = expr | expr .* expr | expr ./ expr | expr .^ expr

type → int | float | bool | string

matrix → mat type | mat type[**lit**] | matrix[**lit**]

actuals → expr | actuals, expr

else_if → else_if elif (expr) stmt | **ε**