Matcat Programming Language Reference Manual

Mariam Khmaladze Davit Barblishvili James Ryan Andreas Cheng

System Architect Language Guru Tester Manager

{mk4069}@barnard.edu,{db3230, james.ryan, hc3142}@columbia.edu

February 24, 2021

Contents

1 Overview of Matcat

The Matcat language is an imperative, mathematically-inclined language whose goal is to aid in the writing and computation of linear algebra operations. Implementing these functions will occur in similar ways to the Wolfram language (see Wolfram: Matrices and Linear Algebra). Matcat aims to mimic functionality and syntax from Java and C.

Our primary goals are:

- To simplify operations on matrices. Examples of these include row reduction, computing transformations, finding eigenvalues/eigenvectors, diagonalizing/transposing a matrix, performing the Gram-Schmidt process to find an orthogonal basis.
- To achieve Java-like or C-like syntax to shorten the learning curve.
- To allow users to write readable code, matrix/vector structures and the operators must mirror mathematical symbols and read intuitively.
- Optimizing compile time with static typing

1.1 Type System Overview

Matcat is strongly typed with a static type scope. A variable must be declared before it is used and can contain only values of the type with which it is declared. The textual structure of a program determines the scope of a variable.

2 Lexical Conventions

2.1 Comments

Matcat has single-line and multi-line comments. Both types of comments do not nest.

All tokens on the line that starts with `//` are part of a single-line comment which are not parsed.

```
// TODO: find column space of m
m = <1,2,3;4,5,6;7,8,9;>
```

Multi-line comments begin with `/*` and end with `*/`.

```
/* Eigenvalues are a set of scalars.
 * We can transform them as eigenvectors.
 *
 *    <- Asteroids between /* and */ are optional.
 *
 */
```

2.2 Identifiers

There are certain rules for defining a valid Matcat identifier. These rules must be followed. Otherwise, a compile-time error arises:

- The only allowed characters for identifiers are all alphanumeric characters([A-Z],[a-z],[0-9]), '$'(dollar sign) and '_' (underscore);
- Identifiers should not start with digits([0-9]). For example, "123cat" is not a valid Matcat identifier;
- Matcat identifiers are case sensitive;
- There is no limit on the length of the identifier, but it is advisable to use an optimum length of 3–15 characters only;
- Reserved words can't be used as an identifier. For example, "int matrix = 20;" is an invalid statement as 'matrix' is a reserved word.

```
// Valid Identifiers
matCat;
thisIsValid;
this_is_valid_too;

//Invalid Identifiers
Mat Cat; //contains space
123cat;  //starts with a digit
2;       // includes only a digit
inverse; //reserved keyword in Matcat
```

2.3 Operators

Matcat utilizes following reserved operators:

```
- +  <  >  <=  >=  !=  ==  *  /  || &&  x  cr ** dot transpose
inverse
```

2.4 Keywords

Matcat has a set of keywords that are reserved words that cannot be used as variables, methods, classes, or any other identifiers:

```
while for if else return true false int float char bool string matrix
vector new print transpose inverse dim det rref
```

2.5 Grouping and Code Blocks

An expression such as x = 0 or x = x + 1 or print(...) becomes a statement when it is followed by a semicolon, as in:

```
 x = 0;
 x = x + 1;
 print(...);
```

In Matcat, the semicolon is a statement terminator rather than a separator as it is in languages like Ocaml. Braces { and } are used to group declarations and statements together into a compound statement or block so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around multiple statements after an if, else, while, or for are another. There is no semicolon after the right brace that ends a block.

2.6 Separators

Matcat uses parentheses to override the default precedence of expression evaluation. To separate expressions and denote control flow Matcat is utilizing ';', {} respectively. It is not possible in Matcat to separate two consecutive expressions on the same line; however, it is possible to declare multiple variables in one line separated by commas and assigning one value to multiple variables is allowed, as well:

```
int myInt = (2+3)*5; // overrides default operator precedence
int a, b, c;         // declaring multiple variables
a = b = c = 5;       // assigning 5 to a, b, and c
```

2.7 Literals

Literals represent strings or one of Matcat's built-in data types: vector, matrix, float, char, int, and boolean.

2.7.1 Float Literals

A float literal is a number with a whole number, an optional decimal point, a fraction, and an exponent.

((([0-9]+\.[0-9]*)|([0-9]*\.[0-9]+))((e|E)(\+|-)?[0-9]+)?|[0-9]+((e|E)(\+|-)?[0-9]+))

Examples of float literals:

```
10
1.2
14.
0.00005
1e+2
2.3E-3
```

2.7.2 String Literals

A string literal is a sequence of characters enclosed in single or double quotation marks, i.e. abcdefghijklmnopqrstuvwxyz. The matching regex is

("[^"'\\]*(\\.[^"\\]*)*")|('[^"'\\]*(\\.[^'\\]*)*')

Example of string literals:

```
"This language is written in Ocaml"
"A string literal"
```

2.7.3 Int Literals

An integer literal is any sequence of integers between 0 and 9. The matching regex is [0-9]+.

2.7.4 Boolean Literals

Boolean types represent true and false. They are represented in Matcat by the **True** and **False**

keywords.

2.7.5 Vector Literals

Vector literal is a one-dimensional array containing the comma-separated arguments as its elements. The type of the resulting vector is automatically determined by the types of arguments inside the braces. If all the arguments are the same type, then that is its type. If the vector contains floats and integers, all the elements are converted to floats.

```
vector vt = [2,3,5,6;] // one-dimensional array representing the
vector; its type is int
vector vt2  = [2,3.2,5,6;] /* since there is one floating point
number in the vector, all the elements of the array are converted to
float [2.0,3.2,5.0,6.0;] */
```

2.7.6 Matrix Literals

Matrix literal is a multi-dimensional array i.e, one array consisting of multiple vectors. All the vectors within the array are treated as Vector Literals (2.7.5). If one of the vectors within the matrix includes floating point numbers, all the elements of all vectors are converted into floating-point numbers.

```
matrix mt = <2,3,4;
             5,6,7;> // a matrix containing two vectors
matrix mt2 = <2.4,3,4;
                5,6,7;> /* Since one of the elements of the vector is
floating point number, all the elements should be converted to
floating point numbers. I.e <2.4,3.0,4.0;
                             5.0,6.0,7.0;> */
```

3 Data Types

Matcat represents all pieces of data as either an object or a primitive. There are only a few basic data types in the language.

3.1 Primitives

Primitives are series of bytes of some fixed length, and there are four primitive types in Matcat:

- ➔ int (4 bytes, 2’s complement);
- ➔ float (8 bytes, IEEE standard double);
- ➔ char (1 byte, ASCII; a single byte, capable of holding one character in the local character set);
- ➔ bool (1 byte; 00000001 for true, 00000000 for false);

  *Note that a vector and matrix are not primitive data types.

3.2 Objects

Any piece of data that can’t be inferred to be one of the primitive types is represented as an object - including vector and matrix. An object has an associated type and an associated value. The value can contain primitives and/or references to other objects. How to interpret the content of the value of an object is determined by its type. References are completely under the hood and not user-accessible as pointers.

3.3 Mutability

The primitive objects in Matcat are immutable, including matrix, vector, ints, floats, and booleans. Strings are not primitives, but are also immutable. All operations which modify these objects actually return new objects. All user defined types, as well as those in the standard library, are mutable, in the sense that modifying them does not overwrite the underlying object. Assigning to any variable will still overwrite its underlying object, i.e. x = 3; x = 4 assigns an integer literal with value 3 to the variable x, and then assigns a different integer literal with the value 4 to the same variable. Likewise, matrix x = <1, 2, 3;>, x = <4, 5, 6;> will assign one object of type list to x, and then assign a different object to it. On the other hand, x = <1, 2, 3;>, x[0][1] = 4 will modify the underlying data associated with the variable x, returning x = <1, 4, 3;>.

3.4 None

None is a keyword that returns a reference to a predetermined object of a special type. There is only ever one object of this type.

3.5 Memory Model

The Matcat language will be ”pass by reference,” as in Java. We are going to implement a memory management system that will be handled internally by a simple garbage collector - objects that no longer have references attached to them deallocated. No explicit memory management is required except for the heap’s object allocation, which is handled by a reserved keyword ‘new’.

3.6 Library Functions

The Matcat language will use some of the library functions that will be accessible to the user for matrix manipulation. The Matcat will include the following functions:

- Dim → will take a matrix as an argument da return two integers describing the dimension of the matrix
- Det → will take a matrix as an argument and return one integer denoting the determinant of the matrix
- Rref → will take a matrix as an argument and return the matrix in rref form using Gaussian elimination.

4 The Matcat Type System

4.1 Overview

The Matcat language uses static typing. All data types need to be set properly at compilation but can change with a properly defined operation in Matcat language such as casting. This guarantees that all the errors will be caught during compilation, and runtime will be error-free.

4.2 Explicit Typing

Explicitly typed variables are denoted by a type like an int, float, string, char, boolean, matrix, vector, followed by a variable name and semicolon.

```
int x = 3;
vector v=[3,4,5;];
string str=”matcat”;
```

Once an identifier has been associated with a class, it cannot be assigned to an object of a different class. Still, the variable can be cast to get the desired result. (Casting is only allowed between integers and floats).

```
int x = 3;
x='s';          //this is not allowed
x=4;          //this is allowed
float y=3.0;
int z=x+(int)y;
```

4.3 Optimization

The compiler infers that a piece of data will consistently be of a certain primitive type, unboxes the data, and converts it to a primitive type. Otherwise, it fails at compile-time. This means that runtime performance will be as optimized as in C and Java.

4.4 Function Typing

Functions in Matcat are declared explicitly. The types of function calls will be checked at a compile-time and will be compiled to efficient machine code. A function is denoted by keyword func, followed by the name of the function, arguments that it takes, return types, and body.

```
func myFunction(dataType x, dataType y) returnType1, returnType2...
{}
```

Arguments and return types of the function must be explicitly typed upon declaration. Matcat allows the return of multiple variables of different data types. Examples will be shown below.

5 Statements and Expressions

5.1 Statements

A Matcat program consists of a series of statements, such as the following:

- Expression
- Class Declaration
- Function Definition
- Return Statement
- If-else statement
- For loop
- While loop

Statements are encapsulated together to form blocks by existing within the same {}.

5.1.1 If-else Statements

An if statement consists of a condition - an expression which evaluates to a Boolean type - and a block of statements which execute if this condition evaluates to true. An else statement optionally follows an if statement and its block of statements execute when its paired if the statement's condition evaluates to false.

```
if(condition) {
   // executes when the condition is true
} else {
   // executes when the condition is false
}
```

5.1.2 While Loops

A While loop consists of a condition and a series of statements. These statements execute until the condition evaluates to false.

```
while(condition) {
   // executes until condition is false
}
```

5.1.3 For Loops

A For statement consists of three parts:

1. An initial assignment expression in which a variable can be defined whose scope is limited to its for loop's block of statements. The type of this variable must be explicitly declared.
2. A condition that must evaluate to true in order to execute the corresponding block of statements.
3. An expression which executes after each loop takes place; typically an increment.

```
for(int i = 0; i < N; i++) {
   // series of statements
}
```

5.2 Expressions and Operators

Expressions consist of statements that chained together using operators. All expressions, even assignments, evaluate to a built-in or user-defined data type.

5.2.1 Unary Operators

Unary operators act on a single variable or expression. These include:

1. Negation

```
int x = -2;
```

2. Boolean Flipping

```
bool a = false;
bool b = !a; // b is true
```

3. Transpose

```
matrix m = <1,2,3; 4,5,6; 7,8,9;>;
matrix t = transpose m; // t holds m transpose
```

4. Inverse

```
matrix m = <1,2,3; 4,5,6; 7,8,9;>;
matrix t = inverse m; // t holds m inverse
```

5.2.2 Binary Operators

Binary operators take the value of the expression immediately to its right and left and reconcile them to produce the value of the overall expression. Both of these expressions must evaluate to legal types for the given operator, detailed below. These follow the form

| Expression * Operator * Expression

where * can represent any of the following operators:

1. Assignment Operator

The assignment operator stores values in variables, with the value on the right stored in the variable on the left. This can be applied to any datatype.

```
string s = "this is stored in variable s";
```

2. Arithmetic Operator
    a. Addition

```
2 + 8 // evaluates to 15
"one " + "two" // evaluates to "one two"
[1, 0] + [0, 1] // evaluates to [1, 1]
```

    b. Subtraction

```
2 - 8 // evaluates to -6
[5, 5] - [4, 2] // evaluates to [1, 3]
```

    c. Multiplication

```
3 * 4 // evaluates to 12
2 * [4, 5, 6] // evaluates to [8, 10, 12]
```

    d. Division

```
9 / 2 // evaluates to float type 4.5
```

    e. Exponentiation

```
3 ** 2 // evaluates to 9
```

3. Vector Specific Operators
    a. Dot Product

```
[2, 3] dot [3, 4] // evaluates to 18
```

b. Cross Product

```
[1, 2, 3] cr [4, 5, 6 ] // evaluates to [-3, 6, -3]
```

4. Relational Operators

```
int x = 1;
int y = 2;
bool b = x < y; // b holds true
b = x == y; // b holds false
b = x != y; // b holds true
```

5.3 Operator Precedence

The table below shows operator precedence, starting from the lowest precedence and increasing down the table.

Operator	Meaning	Associativity
;	Sequencing	Left
=	Assignment	Right
.	Access	Left
||	Logical Or	Left
&&	Logical And	Left
==, !=	Equality, Inequality	Left
<, >, <=, >=	Comparisons	Left
+, -	Addition, Subtraction	Left
*, /	Multiplication, Division	Left
**	Exponentiation	Right
!	Logical Not	Right

5.4 Functions

A function is a type of statement. It takes a list of arguments and returns a list of values. Both arguments and return values must have explicit types and must be passed and caught in the order given in the function definition. The syntax is shown below.

```
func sumAndDiff(int one, int two) int, int {
   return (one + two), (one - two)
}
```

Function sumAndDiff expects two integer arguments, and returns two integer arguments as stated by the datatypes after the ().

5.5 Function Calls

Functions are called using their identifier and passing legal arguments within the (). Return values can be assigned to newly declared or previously existing variables. In the case that a function returns multiple values, the programmer does not need to capture each of them in a variable; an underscore indicates a return value that is passed over and discarded.

```
int diff;
_, diff = sumAndDiff(5, 4); // diff holds 1, sum return value passed over

/* these two lines are equivalent */
int sum = sumAndDiff(1, 2);
int sum, _ = sumAndDiff(1, 2);
```

6 Standard Library

6.1 Lists

Matcat has a built-in list data structure with a dynamic length that can hold objects of arbitrary type. The following operations are supported:

Function / Operators	Behavior
list[n]	Returns the n-th element in the list

6.2 Strings

Strings with the keyword string are implemented with a list of char that are immutable. So, changing a string would result in a new string. The following functions are supported:

Function / Operators	Behavior
str.upcase()	Returns a string with uppercase characters
str.downcase()	Returns a string with lowercase characters
str.capitalize()	Return a lowercase string with each word capitalized.
str[i]	Returns the i-th character in str

6.3 print()

print(variable) sends a string representation of the variable based on its type. Matcat provides built-in string representations for primitive types, vectors, and strings. Whitespaces in strings are not ignored.

Strings also allow interpolation of other values using `#{...}`:

```
string name = "Sdrawde"
print("Hi \#{#{ name }\}.");   //-> Hi #{Sdrawde}.
```

Special characters can be escaped with a backslash.

6.4 Vectors

Matcat has a built-in vector structure with a dynamic length that can hold vectors consisting of integers, floats, or both. Internally, vectors convert all integer elements to float elements.

Vector can be declared with the following syntax:

```
vector v = [1,0,0;];
```

The following operations are supported:

Method/operator	Behavior
v[i]	Returns the i-th element in the vector
v1 + v2	Returns the vector sum of v1 and v2
v1 - v2	Returns the vector difference of v1 and v2
v1 dot v2	Returns the dot product float of v1 and v2
v1 cr v2	Returns the cross product vector of v1 and v2
scalar * v	Returns the vector with scalar multiplication performed
v1 == v2	Returns true if v1 is identical to v2. Otherwise, false.

7.4 Matrix

On top of the vectors, Matcat also provides an out-of-the-box matrix. Matrices are built with vectors that consist of integers, or floats, or both. Vectors that contain elements other than integers or floats are not allowed in a matrix.

Matrix can be declared with the following syntax:

```
matrix m = <1,0,0;
            0,1,0;
            0,0,1;>;
```

The following operations are supported:

Method/operator	Behavior
m[i][j]	Returns the j-th element in the i-th row
m1 + m2	Returns the matrix sum of m1 and m2
m1 - m2	Returns the matrix difference of m1 and m2
m1 * m2	Returns the matrix with matrix multiplication performed
scalar * m	Returns the matrix with scalar multiplication performed
m1 == m2	Returns true if m1 is identical to m2. Otherwise, return false.
transpose m	Returns a new matrix as transpose of m.
inverse m	Returns a new matrix as inverse of m if exists else returns NULL.
rref(m)	Returns matrix in rref form using Gaussian elimination.
dim(m)	Returns two integers (row, col) indicating the dimension of the matrix m.
det(m)	Returns a float determinant value of the matrix m

7 Sample Code

```
//declare 2x3 matrix
matrix m=<1,2,3;0,1,2;>;

//declare a vector
vector v =[3,4,5;];

//transpose matrix, result-3x2 matrix
m=transpose(m);
matrix d=inverse(m);

//dot product
int x=m dot d;
//cross product
matrix m1 = <1,0,0;
             0,1,0;
             0,0,1;>;
matrix m2 = <1,2,3;
             4,5,6;
             7,8,9;>;

// scalar multiplication
m1 = 5 * m1;

matrix cp = m1 cr m2;

//solving systems of linear equations
func sol_lin_equ(matrix a, matrix y) matrix {
   matrix a_inverse=inverse(a);
   matrix solution=a_inverse * y;
   return solution;
}

matrix solution = sol_lin_equ(m1, m2);
print(solution);
```