M^2 Reference Manual

Jeffrey Monahan - jm4155
Christine Pape - cmp2223
Shelley Zhong - sz2699
Tengyu Zhou - tz2338

October 17, 2017

Contents

1 Introduction

1.1 Motivation

Matrices are an integral part of many mathematical operations and have uses ranging anywhere from graphics to cryptography. Some of the operations, however, that can be performed on them can get messy and complicated very quickly when done by hand. By creating this language, we hope to optimize how people work with matrices, saving time both on initial computation and going back to fix potential errors.

1.2 Language Description

The primary goal of our language is to provide a platform to both easily define and manipulate matrices. To this end, we aim to implement the basic building blocks of all complex matrix functions (e.g. cross product, inverse, transpose, etc.) as well as implement an intuitive method to define and manipulate vectors of various sizes. Furthermore, we will implement the fundamental operators needed to perform most mathematical operations. We feel that it is important the program has the ability to interact with the user through I/O, so we keep the idea of strings, printing, and scanning user input. We also keep basic fundamentals, such as conditional statements and loops. Our syntactical style while remain similar to the widely used languages (i.e. We will be keeping our language easy enough to learn by someone who already knows modern programming languages while adding enough matrix-specific features so that it is specialized.

Ultimately, our programming language is designed to be used as a tool for manipulating matrices, so the types of programs meant to be written in this language reflect that. Besides performing basic operations, programmers should be able to build programs that can solve systems of equations, encrypt/decrypt messages, or even create graphs and images.

2 General Syntax

Comments

Comments have two different forms: single line comments and block comments.

- A single line comment begins with characters //
- A block comment begins with characters /* and end with characters */

Identifiers (data variables) $\Sigma : \{A - Z, 0 - 9, \_\}$

Identifier of data variables is a sequence of characters consists of only uppercase letters, numbers, and underscores. The first character must be an uppercase letter.

Identifiers (math variables) $\Sigma : \{a - z\}$

Identifiers of math variables is a sequence of characters consists of only lowercase letters. These identifiers can only be used within an equation.

Reserved words

The following identifiers are reserved for keywords, and should not be declared or used in other circumstances.

- `if`
- `else`
- `return`
- `while`
- `for`
- `switch`
- `break`
- `continue`
- `void`
- `int`
- `char`
- `float`
- `double`
- `default`
- `new`
- `main`
- `struct`
- `matrix`
- `string`

Grouping

{ }

Braces are used for grouping a bunch of statements into a single unit. Nested groupings are allowed, but the characters '{' and '}' should always be balanced.

‘ ’

Single characters will be enclosed by ‘ ’

“ ”

Strings will be enclosed by “ ”

Terminator

;

Every statement ends with the character ';'. Users should always include ; at the end of statement even if the statement is the last statement in a { } group.

Character Constants

The following sets of characters in single quotes ‘ ’ will be used to define special characters and whitespace as follows:

- \n newline
- \r carriage return
- \t tab
- \’ ’

- \" "
- \\ \

An example of their use is

```
string x = "She said \"hello.\"\n";
```

The value of x would then be "She said "hello." followed by a new line.

Variable Definition

Variables will be defined as **typename VAR_NAME**
For example, to define an integer called count, one would type:

```
int COUNT;
```

The initialization of variables at declaration is optional.

White Space

Whitespace is ignored unless it is included as a char or string literal using " "

3 Data Types

3.1 Primitive Data Types

int

Standard representation of signed or unsigned integers. Example declaration:

```
int X = 1;
```

Literal:

```
(-|ϵ)(1-9)(0-9)*
```

float

Standard representation of signed or unsigned single-precision floating point numbers. Example declaration:

```
float X= 1.0;
```

Literal:

```
(-|ϵ)(0-9)+.(0-9)+
```

double

Double-precision floating point number. Example declaration:

```
double X  = 1.0;
```

Literal:

```
(-|ϵ)(0-9)+.(0-9)+
```

char

16 bit datatype representation. Example declaration:

```
char C = 'a';
```

Literal:

```
\'(\s|\w|\W|ϵ)\'
```

boolean

Standard representation (holds values 1 or true, for true, and 0 or false, for false. Example declaration:

```
bool X  = false;
```

Literal:

```
((false|true)|(0|1))
```

3.2 Supported Data Types

Matrix

Storing a matrix is not all that different to storing a multidimensional array. However, by implementing it as it's own data type, it provides an easy interface for users to work with and manipulate that would be much more difficult if they had to access each element individually. For example, calculating a cross product of two 3×1 matrices involves six separate calculations. That doesn't sound like a lot, but with such a common operation, it would be much easier to simply write $A \times B$ rather than accessing the elements of A and B individually and storing the results in a new matrix. Many operations can be simplified for the user in this way.

The Matrix type has a simple definition similar to that of an array in Java. An matrix is stored as an array of elements, where the top left is the 0^{th} element and the bottom right is the last. Example declaration for a 2×3 matrix:

```
Matrix<int> M = new Matrix<int>[2, 3];
```

Matrix literals take the form of

$$\{(x_1, ..., \ x_n), ...(y_1, ..., \ y_n)\}$$

where $x_1, x_2, ..., \ x_n$ and $y_1, y_2, ... \ y_n$ all share a common type. For example,

```
Matrix<type> M = {(x, y, z), (a, b, c)};
```

The line above will create a 2×3 Matrix of the form

$$\begin{bmatrix} x & y & z \\ a & b & c \end{bmatrix}$$

where x, y, z, a, b, c are instances of <type>. However, ill-formatted definitions such as

```
Matrix M = {(a,b,c),(x,z)};
```

would throw a syntax error because the number of columns is mismatched.

string

Strings are stored as arrays of characters. Example declaration:

```
string A = new string("hello world");
```

Literal:

```
\"(\s|\w|\W)*\"
```

array

Arrays are collections of data with the same type. The array size cannot be changed after initialization, and each entry can be indexed by $A[i]$, where A is the array name and i is an integer between 0 and (arraysize -1). Arrays can be nested, and multi-dimensional arrays are permitted as well. Example declaration:

```
int[] A = new int[10];
```

Alternative declaration:

```
int[] B = {1, 2, 3, 4};
```

Literal:

```
\{((0-9),|((1-9)(0-9)+,))+(0-9)|(1-9)(0-9)+\}
```

3.3 Declarative Statements

Declaring a variable for primitive types involves a general syntactical structure of:

```
variable_type variable_name;
```

or

```
variable_type variable_name = value;
```

In the case of a non-primitive data type (e.g. string), the variable is instantiated according to it's unique syntactical form. These are defined in section 3.2. For example

```
string Hello;
```

or

```
string Hello = new string ("hello");
```

4 Operators

4.1 Basic Operators

Arithmetic Operators

$$+, -, *, \times, /, \%, \hat{}$$

Relational Operators

$$==, !=, >, <, >=, <=$$

Logical Operators

$$!, \&\&, ||$$

Assignment Operator

$$=$$

4.2 Matrix-Specific Operators

Matrix and scalar multiplication *

Example:

```
A * B
```

▷ This will alter the matrix B, according to multiplication by the scalar A, or it will produce a new matrix C, which is the result of the multiplication of the two matrices A and B

▷ This will throw an error if the user attempts to multiply two matrices with non-compatible dimensions.

Scalar division /

Example:

```
A / N
```

▷ This will alter the matrix A according to division by the scalar N.

Power ^

Example:

```
A ^ B
```

▷ This will output a new matrix C by producing the result of the matrix multiplication of B copies of A.

▷ This will throw an error if the user attempts to use any data type other than int, as well as if B is negative (in order to manipulate the inverse, the function invert(Matrix A) must be used).

Dot product ·

Example:

```
A · B
```

▷ This will output a new matrix C, which is the dot product of A and B.

Cross product ×

Example:

```
A × B
```

▷ This will output a new matrix C, which is the cross product of A and B.

5 Conditionals, Loops

5.1 Conditionals

If statement

The four forms of the conditional statement are

```
if (expression) {statement1}
```

▷ if expression is non-zero, statement1 will be executed

```
if (expression) {statement1}
else {statement2}
```

▷ if expression is non-zero, statement1 will be executed
▷ If expression is zero, statement 2 will be executed

```
if (expression1) {statement1}
else if (expression2) {statement2}
```

▷ if expression1 is non-zero, statement1 will be executed
▷ If expression1 is zero and expression2 is non-zero, statement 2 will be executed.
▷ Multiple **else if** clauses are allowed following an **if** clause. Expressions will be evaluated from left to right and until one is non-zero and its corresponding statement is executed. Once a statement is executed, no more expressions in this set will be tested.

```
if (expression1) {statement1}
else if (expression2) {statement2}
...
else {statement3}
```

▷ if expression1 is non-zero, statement1 will be executed
▷ If expression1 is zero and expression2 is non-zero, statement 2 will be executed.
▷ Multiple **else if** clauses are allowed following an **if** clause. Expressions will be evaluated from left to right and until one is non-zero and its corresponding statement is executed. Once a statement is executed, no more expressions in this set will be tested.
▷ If all expressions have evaluated to zero (meaning no statement was executed, statement3 will be executed.

5.2 Loops

While Statement

The while statement has the form of

```
while (expression) {statement}
```

The expression is checked first, and whenever it's non-zero, the statement will be executed. After that it will return back to check the expression and repeat until the expression is zero.

For Statement

The for statement follows the syntactical form of

```
for(index_variable declaration/instantiation;
   conditional_statement; action) {statement}
```

Before the first iteration, the index variable will be created/initialized. The for statement will then check the conditional statement. If the conditional statement is evaluated to be non-zero, the statement will be executed. Upon completing the statement, the action will be executed. The conditional statement will once again be evaluated. This process will repeat until the conditional statement evaluates to zero.

6 Functions

6.1 Built-in Functions

Print Functions

There are two print functions.

- `print(string s);`

prints s to the console. If s should include variable values, they can be formed as following:

```
"" + var + ""
```

- `println(string s);`

is defined the same as above, except it automatically places a newline after the string.

Data type conversion functions

There are several functions to convert data types.

- `intToString(int x);`
  - ▷ Convert an int to a string.
- `floatToString(float x);`
  - ▷ Converts float to a string.
- `doubleToString(double x);`
  - ▷ Converts double to a string.
- `stringToInt(string s);`
  - ▷ Converts string to an int.
- `stringToFloat(string s);`
  - ▷ Converts string to a float.
- `stringToDouble(string s);`
  - ▷ Converts string to a double.

Matrix specific functions

The following functions are applied to matrices.

- `determinant(Matrix A)`
  - ▷ Calculates determinant of A
  - ▷ Returns a signed integer

- `transpose(Matrix A)`

  ▷ Calculates transposed matrix of A.

  ▷ Returns the transpose of A in a copy, that is, A is not modified.

- `inverse(Matrix A)`

  ▷ Calculates the inverse matrix of A

  ▷ Returns the inverse of A in a copy, that is, A is not modified.

- `identity(int dimension)`

  ▷ Returns an identity matrix of a size corresponding to the given dimension parameter. For instance,

```
identity(3)
```

will return a matrix of the form

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

- `copy(Matrix A)`

  ▷ Returns a copy of A

- `columns(Matrix A)`

  ▷ Returns the number of columns in A

- `rows(Matrix A)`

  ▷ Returns the number of rows in A

- `makeMatrix(Matrix A, int rows, int columns)`

  ▷ Returns a matrix from an array with specified dimensions

- `size(array A)`

  ▷ Returns the size of array A.

6.2 User-defined Functions

A user can write their own functions to be called in their programs. The format for creating these functions is:

```
return_type function_name(param_type1 param_name1, ...
param_typeN param_nameN){statement}
```

• return_type: type of variable the function will return. Void if none.

• function_name: the name of the function. It must start with a letter, but the alphabet is defined as $\Sigma : \{A - Z, a - z, 0 - 9, \_\}$

▷ Functions cannot be overloaded. That is, two functions cannot have the same name even if their parameter list differs.

• param_type1 param_name1, ... param_typeN param_nameN: List of parameters to be sent into the function. The type for each parameter must be identified as well as the name they will be referenced by within the function.

• statement: the code to be executed when the function is called

All functions must be defined before they can be called.
All code to be executed will be defined in the main() function.
Sample code for calculating GCD:

```
int gcd(int A, int B) {
  int C;
  while(B != 0) {
    C = A % B;
    A = B;
    B = C;
  }
  return A;
}
```

7 Structures

M^2 does not support classes, but does provide structures for grouping data.
A structure can be defined as such:

```
struct structname {
vartype data1;
...
vartype dataN;
};
```

Once defined, an instance of a structure can be created by:

```
struct struct_name VAR_NAME;
```

And its data can be accessed by:

```
VAR_NAME.data1
...
VAR_NAME.dataN
```

8 Scope

8.1 Scope of Variable

Global Variables

Global variables are defined outside of main or any function. They can access by the entire program.

Local Variable

Local Variables are defined within main or a function. Copies of local variables can be passed to functions.

Scope of local variables is within the { } brackets in which they are created. For instance, if variable is created in a for loop, it would be deallocated after the for loop has executed.

8.2 Scope of functions

New user-defined functions have to defined before they are called.

9 Library

Scanner allows for user input to be implemented within a program's structure. Scanning only has the capability to read in input from the keyboard and therefore does not include an input stream as a parameter. The scan function waits for user input by watching for the user to input the return key. Return keys are ignored in the line of user input. The return type of the scan() function is a string of the entire line of user input.

```
string input = scan();
```

10 References

[1] Dennis M. Ritchie, "C Reference Manual," https://www.bell-labs.com/usr/dmr/www/cman.pdf. Web. Oct 15, 2017.

[2] D. Watkins, E. Chen, P. Schiffrin, K. Atef, "JFlat Reference Manual," http://www.cs.columbia.edu/sedwards/classes/2015/4115-fall/lrms/Dice.pdf. Web. Oct 15, 2017.