MATRX LRM

Co-Manager: Katie Pfleger (kjp2157)
Co-Manager: Julia Sheth (jns2157)
Language Guru: Alana Anderson (afa2132)
System Architect: Pearce Kieser (pck2119)
Tester: Nicholas Sparks (ns3284)

0 Introduction

Machine learning (ML) is the area of computational science that focuses on analyzing and interpreting patterns and structures in data in order to enable learning, reasoning, and decision making. While the use of machine learning has been around since the turn of the century, it has only recently become mainstream in the industry. Today, 51% of enterprises across a variety of industries are deploying machine learning in production. In fact, job titles such as “machine learning engineer,” “deep learning engineer,” and “data scientist” are already widely used terms. As these engineers will tell you, though, machine learning really boils down to one thing: *matrices*.

A matrix is a two-dimensional array of scalars with one or more columns and one or more rows. Matrix manipulations are often essential to machine learning algorithms, where they are used as the input data when training algorithms. However, implementing these operations in common programming languages (such as C, C++, or python) can be extremely complicated and time-consuming. While libraries and tools with more robust matrix manipulation tools exist, they are often expensive and syntactically complex. With this motivation, we have decided to build a simple language that supports matrix operations by design.

1 Lexical Elements

This chapter describes the lexical elements that make up MATRX source code after processing. We refer to these elements as tokens. We specify five types of tokens: keywords, identifiers, constants, operators, and separators.

1.1 Comments

The characters /* introduce a comment, which terminates with the characters */.

1.2 Identifiers

Identifiers are sequences of characters used for naming variables and functions. Users may use letters and the underscore character '_' in identifiers. Identifiers are case sensitive, such that foo and FOO are two different identifiers.

1.3 Keywords

Keywords are special identifiers reserved for use as part of the programming language itself. In MATRX, we have the following keywords:

```
break, char, col, continue, double, else, float, for, if, int, matrix, return, row, void, while
```

1.4 Constants

1.4.1 Integer Constants

An integer constant is a sequence of digits, with an optional prefix to denote the base. We use the prefixes '0x' to indicate hexadecimal, '0' to indicate octal, and no prefix to indicate decimal.

Here are some examples:

```
/* hexidecimal constants */
0xAB42
0x88
0x1

/* octal constants */
057
012
03

/* decimal constants */
2018
12
8
```

1.4.2 Character Constants

A character constant is usually a single character enclosed within quotation marks, such as ‘A’. A character constant is of type ‘int’ by default. Some characters cannot be expressed using only one character. To represent such characters, we use the following:

- \\ backslash
- \’ single quotation mark
- \” double quotation mark
- \b backspace
- \n newline
- \t horizontal tab
- \v vertical tab

1.4.3 Real Number Constants

A real number constant is a value that represents a fractional number. It consists of a sequence of digits which represent the integer, a decimal point, and a sequence of digits which represent the fraction.

Here are some examples:

```
4.7
4.
4
.7
0.7
```

1.4.4 String Constants

A string constant is a sequence of zero or more characters, digits, and escape sequences enclosed within double quotation marks. A string constant is a one-dimensional matrix of chars, where the elements of the matrix are characters (see 2.5). All string constants contain a null termination character (\0) as their last character to indicate the end of the string.

Here are some examples:

```
/* a simple string constant */
"matrix languages are the best languages"
```

1.5 Operators

An operator is a special token that performs an operation. Full coverage of operators can be found in Chapter 3 of this Language Reference Manual.

1.6 Separators

A separator separates tokens. White space is a separator, but not a token. We have the following separators:

```
( ) [ ] { } ; , . :
```

1.7 White Space

White space is the collective term used for several characters: the space character, the tab character, the newline character, the vertical tab character, and the form-feed character. White space is ignored (outside of string and character constants), and is therefore optional, except when it is used to separate tokens.
This means that:

```
#include <stdio.h>
int main()
{
  printf( "hello, world\n" );
  return 0;
}
```

is functionally the same as:

```
#include int main() { printf( "hello, world\n" ); return 0; }
```

White space is not required between operators and operands, nor is it required between other separators and that which they separate.
This means that:

```
x++;
matrix m = [ [0, 1]
             [2, 3] ];
```

is equivalent to:

```
x ++;
matrix m = [[0, 1][2, 3]];
```

In string constants, spaces and tabs are included in the string.
This means that:

```
"This is a string with spaces."
```

Is **not** the same as:

```
"Thisisastringwithspaces."
```

2 Data Types

2.1 Matrices

A matrix is a data structure that lets you store a two dimensional array of numbers. A matrix has at least one row and at least one column.

2.1.1 Declaring Matrices

Matrices can be declared by specifying an the identifier, the number of rows, and the number of cols. Note that the type of data stored is not specified until the matrix is initialized (see 2.1.2).

Here is an example:

```
matrix m[1][3]; /* declares a matrix with 1 row and 3 col */
```

2.1.2 Initializing Matrices

You can initialize elements in a matrix when you declare it by listing each row as a list of elements separated by commas and enclosed by square braces. The data type contained by a matrix and the number of rows and cols is determined when it is initialized. Note that white space does not change the initialization (see 1.7).

Here is an example:

```
matrix a = [ [1, 2] [3, 4] ]; /* declares a matrix [1 2] */
                              /*                  [3 4] */

matrix b = [ [1, 2]
             [3, 4] ]; /* declares the same matrix as above */
```

When a matrix is declared with an incompatible number of rows and cols, we throw an error:

```
matrix m[1][3];

m = [ [1]
      [3, 4] ]; /* this will throw an error */

m = [ [1]
      [3, 4]
      [5, 6]; /* this will throw an error */
```

2.1.3 Manipulating Matrices

We provide operators for many mathematical operations on matrices (see 3.5). We also provide several built-in functions:

- cols(): returns the number of columns in the matrix
- rows(): returns the number of rows in the matrix
- elements(): returns the total number of elements in the matrix
- empty(): returns 1 if the matrix has not been initialized
- get(int row, int col): returns the element at [row][col]
- set(int row, int col, NEW ELEMENT): sets the element at [row][col]
- printm(): prints the matrix

Note that the operations cols(), rows(), elements(), get(int row int col), set(int row, int col), type(), and printm() will throw an error if the matrix that is being called on has not been initialized.

Here are some examples:

```
matrix a = [ [1, 2]
             [3, 4] ]; /* declares a matrix [1 2] */
                       /*                   [3 4] */

int col = a.cols(); /* col is 2 */
int row = a.rows(); /* row is 2 */
int size = a.elements(); /* size is 4 */
```

```
int x = a.get(0, 0); /* x is 1 */

a.empty(); /* returns false */

a.set(0, 0, 2); /* a is now [2 2] */
                /*          [3 4] */

ptinfm(a); /* prints the matrix as [2 2] */
           /* .                   [3 4] */
```

Note that matrices are mutable, such that manipulating an element will change the object in memory.

2.1.4 Multi-dimensional Matrices
We will allow for multi-dimensional matrices to be created, as long as all of the matrices within the overall matrix hold the same type of data.

Note that uninitialized matrices cannot be added to another matrix, as they will not yet have an assigned type, so it will be unknown whether or not their type matches the type of the other matrices within the matrix.

Here are some examples:

```
matrix a = [ [ [ [ 1, 2 ] [ 3, 4 ] ] ]  [ [ [ 1, 2, 3 ] [ 4, 5, 6 ] ] ] ];
matrix b = [ [ 5, 6 ] , [ 7, 8 ] ];
matrix c = [ [ 9, 10, 11] [ 12, 13, 14 ] ];
matrix d = [ [ c ] [ b ] ];

matrix e = [ [ matrix m[1][3] ] [ matrix n [2][4] ];
/* this will throw an error because the matrices m and n have not been initialized */
```

```
matrix f = [ [ [ [ 1, 2 ] [ 3, 4 ] ] ]  [ [ [ 1.0, 2.5, 3.6 ]
[ 4.7, 5.8, 6.3 ] ] ] ];
/* this will throw an error because the two matrices in f are
of different types */
```

2.2 Integers

Integer types can be used for storing whole number values. We support a 32-bit int data type, which can hold integer values in the range of −2,147,483,648 to 2,147,483,647.

Here are some examples of declaring and defining integer variables:

```
int a;
int a = 10;
```

2.3 Floats

The float data type's minimum value is stored in the FLT_MIN, and should be no greater than 1e-37. Its maximum value is stored in FLT_MAX, and should be no less than 1e37.

2.4 Doubles

The double data type is at least as large as the float type, and it may be larger. Its minimum value is stored in DBL_MIN, and its maximum value is stored in DBL_MAX.

Here are some examples of declaring and defining floating point and double variables:

```
double d;
double d = 3.14;
float f;
float f = 10.0;
```

2.5 Chars

A char object may be used anywhere an int may be. In all cases the char is converted to an int by propagating its sign through the upper 8 bits of the resultant integer.

Here are some examples:

```
char w = '1';
char y = 'B';
```

We allow users to specify a string as a one-dimensional matrix of chars, where the elements of the matrix are characters. All string constants contain a null termination character (\0) as their last character to indicate the end of the string.

We provide a built-in function for printing strings, printf. It accepts a string (one-dimensional matrix of chars) as first argument, and prints subsequent arguments according to specifications contained in this format string. Most characters in the string are simply copied to the output; two-character sequences beginning with ‘‘%’’ specify that the next argument should be printed in a style as follows:

- %d decimal number
- %o octal number
- %c ASCII character, or 2 characters if upper character is not null
- %s string (null-terminated array of characters)
- %f floating-point number

Here are some examples:

```
matrix hi = [['h', 'i', '\0']]; /* creates string "hi" */
printf("The magic word is %s.", hi); /* prints "The magic word is hi."
```

3 Expressions and Operators

3.1 Expressions

An expression consists of at least one operand and zero or more operators. An operand is defined as a typed object such as a constant, variable, or function call that returns a value. An operator specifies an operation to be performed on the operand(s).

Here are some examples:

```
42
2 + 2
```

We let parentheses group subexpressions. Innermost expressions are evaluated first. In the example below, (3 + 10) is evaluated to 13 and (2 * 6) is evaluated to 12. Then, 12 is subtracted from 13. Finally, the result of that subtraction, 1, is multiplied by 2.

```
(2 * ((3 + 10) - (2 * 6)))
```

3.2 Assignment Operators

Assignment operators store values in variables. The standard operator = stores the value of its right operand in the variable specified by its left operand. The left operand cannot be a literal or constant.

Here are some examples:

```
int x = 10;
float y = 41.1 + 0.9;
```

Compound assignment operators perform an operation involving both the left and right operands, and then assign the resulting expression to the left operand. Here is a list of the compound assignment operators, and a brief description of what they do:

- += adds the two operands together, and then assign the result of the addition to the left operand
- -= subtract the right operand from the left operand, and then assign the result of the subtraction to the left operand
- *= multiply the two operands together, and then assign the result of the multiplication to the left operand
- /= divide the left operand by the right operand, and assign the result of the division to the left operand
- %= perform modular division on the two operands, and assign the result of the division to the left operand

3.3 Incrementing and Decrementing

The increment operator ++ adds 1 to its operand. The operand must be a either a variable of one of the primitive data types. You can apply the increment operator either before or after the operand.

Here are some examples:

```
int x = 1;
```

```
x++; /* x is now 2 */


char y = 'A';
++y; /* y is now 'B' */
```

3.4 Arithmetic Operators

We provide operators for standard arithmetic operations: addition, subtraction, multiplication, and division, along with modular division and negation.

Here are some examples:

```
a = 5 + 3;
b = 43.5 - 1.5;
c = 5 * 10;
d = 35 / 5;
e = 78 % 26;
f = -5;
```

3.5 Matrix Operators

We support many mathematical operations on matrices, including determinant, dot product, transpose, inverse, scalar multiplication, matrix addition, and matrix multiplication. Note that these operations are only supported for matrices of int, float, double, or char types. When such operations are called on a matrix that stores some other data type, we throw an error.

Here are some examples:

```
matrix a = [[1, 0, 1]];
matrix b = [[0, 1, 0]];
matric c = [[0]
            [1]
            [0]];

matrix d = a + b; /* c is [[1, 1, 1]] */
matrix e = a * c; /* e is [[0]] */
```

3.6 Comparison Operators

Comparison operators can be used to determine how two operands relate to each other (i.e. equal, one less than the other, one greater than the other). The result of these expressions is 1 if the expression is true and 0 if the expression is false.

```
/* equal-to operator */
if (x == y)
  printf('x is equal to y');
else
  printf('x is not equal to y');

/* not-equal-to operator */
if (x != y)
  printf('x is not equal to y');
else
  printf('x is equal to y');

/* less-than operator */
if (x < y)
  printf('x is less than y');
else
  printf('x is greater than y');

/* greater-than operator */
if (x > y)
  printf('x is greater than y');
else
  printf('x is less than y');
```

Note that elements of a matrix can be accessed using the built-in get() function and compared if the data type stored is of type int, float, double, or char. If elements of matrices that store different types are compared, we throw an error.

Here are some examples:

```
matrix a = [[1, 0, 1]];
matrix b = [[0, 1, 0]];

if (a.get(0, 0) == b.get(0, 0))
  printf('this will not print');
else
  printf('this will print because a[0][0] is 1 and b[0][0] is
0');

matrix c = [[1.0, 2.0, 3.0]];
matrix d = [[1, 2, 3]];

if (c.get(0,0) == d.get(0,0))
  printf('this will throw an error'); /* throw an incompatible
data type comparison error */
```

3.7 Logical Operators

Logical operators test the truth value of a pair of operands. Any nonzero expression is considered true, while any expression that evaluates to zero is false. We use && to test if two expressions are both true and || to determine if at least one of two expressions is true. We prepend a ! to flip the truth value of an expression.

```
/* the logial and operator */
if ((x == 1) && (y == 2))
  printf('x is 1 and y is 2');

/* the logical or operator */
if ((x == 1) || (y == 2))
  printf('either x is 1 or y is 2');
```

```
/* the logical negator */
if (!(x == 1))
  printf('x is not 1')
```

3.8 The Comma Operator

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. It should be avoided in situations where comma is given a special meaning, for example in actual arguments to function calls and lists of initializers.

Here is an example:

```
x++, y = x * x;
```

3.9 Operator Precedence

The following is a list of types of expressions, presented in order of highest precedence first. Sometimes two or more operators have equal precedence; all those operators are applied from left to right unless stated otherwise.

1. function calls
2. unary operators (including logical negation, increment, decrement, unary positive, unary negative, indirection operator, address operator, type casting, and sizeof expressions)
3. multiplication, division, and modular division expressions (including matrix operations of these types)
4. addition and subtraction expressions (including matrix operations of these types)
5. greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to expressions
6. equal-to and not-equal-to expressions
7. logical AND expressions
8. logical OR expressions
9. conditional expressions
10. all assignment expressions, including compound assignment
11. comma operator expressions

4 Statements

Except as indicated, statements are executed in sequence.

4.1 Expression statement

Most statements are expression statements, of the form:

```
expression;
```

Usually expression statements are assignments or function calls.

4.2 Conditional statement

The two forms of the conditional statement are:

```
if ( expression ) statement
if ( expression ) statement else statement
```

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the ‘‘else’’ ambiguity is resolved by connecting an else with the last encountered else-less if.

4.3 While statement

The while statement has the form:

```
while ( expression ) statement
```

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

4.4 For statement

The for statement has the form:

```
for ( expression-1opt ; expression-2opt ; expression-3opt ) st
atement
```

This statement is equivalent to:

```
expression-1;
while ( expression-2 ) {
statement
expression-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression typically specifies an incrementation which is performed after each iteration. Any or all of the expressions may be dropped. A missing expression-2 makes the implied while clause equivalent to ‘‘while(1)’’; other missing expressions are simply dropped from the expansion above.

We include the following specialized for statements to iterate through matrices, using the keywords ‘row’ and ‘col’.

```
matrix a = [ [1, 2]
             [3, 4]
             [5, 6] ]; /* declares a matrix [1 2] */
                       /*                   [3 4] */
                       /*                   [5 6] */

for (row r in a)
  r *= 2; /* multiplies each row of matrix a by 2 */
             /* a is now [2 4]                       */
             /*          [6 8]                       */
             /*          [10 12]                     */

for (row r in a) {
    for(col c in a) {
        printf("The number is %d\n", a.get(r, c))
    }
} /* prints "The number is 2"
            "The number is 4"
            "The number is 6"
            "The number is 8"
            "The number is 10"
            "The number is 12" */
```

4.5 Break statement

The break statement causes termination of the smallest enclosing while or for statement; control passes to the statement following the terminated statement.

4.6 Continue statement

The continue statement causes control to pass to the loop-continuation portion of the smallest enclosing while or for statement; that is to the end of the loop.

4.7 Return statement

A function returns to its caller by means of the return statement, which has one of the forms:

```
return ;
return ( expression ) ;
```

In the first case no value is returned. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

4.8 Null Statement

The null statement has the form

```
;
```

A null statement is useful to carry a label just before the ‘‘}’’ of a compound statement or to supply a null body to a looping statement. In the following example, a null statement is used as the body of the loop:

```
for (i = 1; i*i < n; i++)
  ;
```

5 Functions

We allow users to define functions to separate parts of a program into distinct subroutines. To write a function, you must create a function definition. Every program requires at least one function, the main function, where the program’s execution begins (see 5.5).

5.1 Function Declarations

You write a function declaration to specify the name of a function, a list of parameters, and the function's return type. A function declaration ends with a semicolon. You should write the function declaration above the first use of the function. Function declarations have the form:

```
return-type function-name (parameter-list);
```

The return type indicates the data type of the value returned by the function. A function that does not return any data type has the return type void. The function name can be any valid identifier. The parameter list consists of zero or more parameters, separated by commas. A single parameter consists of a data type and an identifier.

Here is an example:

```
int add(int a, int b);
```

5.2 Function Definitions

A function definition specifies what the function does. Function definitions must specify the name of a function, the list of parameters, the return type, and the body of the function. Function definitions have the form:

```
return-type function-name (parameter-list)
{
  function-body
}
```

Here is an example:

```
int add(int a, int b)
{
  return a + b;
}
```

5.3 Calling Functions

Functions are called by using its name and supplying the necessary parameters. Function calls have the form:

```
function-name (parameters)
```

A function call can make up an entire statement or be used as a subexpression:

```
/* as an entire statement */
foo (5);

/* as a subexpression */
x = foo (5);
```

5.4 Function Parameters

Function parameters can be any expression—a literal value, a value stored in variable, an address in memory, or a more complex expression built by combining these. Within the function body, the parameter is a local copy of the value passed into the function; you cannot change the value passed in by changing the local copy.

Here is an example:

```
int foo (int a)
{
  a = 2 * a;
  return a;
}

int x = 42;
foo (x); /* does not change the value of x */
x = foo (x); /* does change the value of x */
```

In the example above, even though the parameter ‘a’ is modified in the function ‘foo’, the variable ‘x’ that is passed to the function does not change when ‘foo (x)’ is called. The original value of x is only changed when we reassign ‘x = foo (x)’.

5.6 The Main Function

Every program requires at least one function, called ‘main’. This is where the program begins executing. The main function does not need a declaration, but must

be defined.

The return type for main is always ‘int’. You do not have to specify the return type for main; however, you cannot specify that it has a return type other than ‘int’. In general, the return value from main indicates the program’s exit status. A value of zero or EXIT SUCCESS indicates success and EXIT FAILURE indicates an error. Otherwise, the significance of the value returned is implementation-defined.

The ‘main’ function can be written to accept no parameters or to accept parameters from the command line. To accept parameters from the command line, the function must have two parameters: argc (an int specifying the number of command line arguments) and argv (a one-dimensional matrix of parameters).

Here are some examples:

```
/* main function with no arguments */
int main ()
{
  printf ("Hello World!");
  return 0;
}

/* main function with command line arguments */
int main (int argc, matrix argv)
{
  int i;
  for (i = 0; i < argc; i++)
    printf ("%s\n", argv[i]);
  return 0;
}
```

6 Scope

Scope refers to what parts of the program can “see” a declared object. A declared object can be visible only within a particular function, or within a particular file, or

may be visible to an entire set of files by way of including header files and using extern declarations. Unless explicitly stated otherwise, declarations made at the top-level of a file (i.e., not within a function) are visible to the entire file, including from within functions, but are not visible outside of the file. Declarations made within functions are visible only within those functions. A declaration is not visible to declarations that came before it.

Here are some examples:

```
int x = 5;
int y = x + 10; /* this will work because x is already defined
*/


int x = y + 10; /* this will not work because y has not yet be
en defined */
int y = 5;
```

7. Nice To Haves

We plan to have file reading into matrices as part of our nice-to-haves. Adding this feature would enable a user to feed data from CSVs or other file types in order to initialize matrices to perform machine learning training.