YAMML: Yet-Another-Matrix-Manipulation-Language: Language Reference Manual

Name: Bill Chen, Janet Zhang, James Xu, Kent Hall, Doria Chen
Uni: gc2677, jz2896, jrx2000, kjh2166, dc3267

Feb 10, 2021

Contents

1 Introduction

The rise of machine learning has seen a rise in need of matrix based computations. Neural Networks used in deep learning, in essence, boils down to a series of matrix operations. Hence, a language that simplifies matrix operations can drastically simplify the workflow ML engineers and architects, allowing for fast prototyping, ease of use, and high likelihood of correctness. With that goal in mind, we aim to create an imperative language that has a familiar syntax to existing languages such as java and C++, while adding built-in support for matrix creating and a series of common matrix operations.

2 Lexical Conventions

2.1 Comments

YAMML has both single-line and multi-line comments. The style is identical to that of C's: Any token followed by // are considered part of a single-line comment and are not lexed or parsed.
Multi-line comments being and end with /* and */

2.2 Identifiers

Valid identifiers are made from ASCII letters and decimal digits. An identifier must begin with a letter, can contain and underscore, cannot be a YAMML keyword.

```
// Valid identifiers
thisisfine
this_is_fine
This_is_FINE
FINE

// invalid identifiers
123id
0
```

2.3 Operators

YAMML uses the following operators

```
+ - ! = ; . < > <= >= != == * / ^ .* ./ && || [a,b] [i:j, k:l]
// Note that + - when used on matrices does point-wise addition and subtraction.
// * is matrix multiplication when used on matrices, .* is element-wise product for matrices,
// ^ is also matrix power, ./ is element-wise matrix division
// || && are logic operators [a,b] and [i:j, k:l] are a matrix operators that index and slice a mat
```

2.4 Keywords

Keywords are reserved identifiers, They cannot be used as ordinary identifiers for other purposes. YAMML Keywords are:

```
if else for while return import
pass continue break print int str bool float
```

2.5 Brackets

YAMML uses curly brackets {} to determine the grouping of statements. This behavior is identical to C. A grouping of statement forms a compound statement, which can be used where one is expected. A compound statement will have the form:

```
compound-statement:
    { statement-list }
statement-list:
    statement
    statement statement-list
```

2.6 Separators

YAMML uses parentheses to override the default precedence of expression evaluation. Parentheses are also used to separate conditions in loops. Semicolons separate expressions, which are the simplest of statements;

```
3 + (5 - 6);

for (int x = 0; x < 10; x += 1){
    print(x);
}
```

2.7 Literals

Literals represents strings or one of YAMML's primitive types: float, char, int, boolean and strings

2.7.1 Float Literals

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

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

// Examples
12
1.2
0.14159
12.
1e+3
12.5e-5
```

2.7.2 Char Literal

A char literal contains only one character surrounded by single quotes assigned to a variable of type char

```
char x;
x = 'a';
```

2.7.3 String Literals

A string literal is a sequence of characters enclosed in double quotation marks

```
(".*[^\\]")
"This is also a string literal"
```

2.7.4 Int Literals

Int literals are a sequence of integers between 0 and 9.

```
[0-9]+
```

2.7.5 Boolean literals

Boolean literals use True or False keywords to represent true or false values.

3 Data Types

3.1 Primitives

Primitives are types of fixed length. The 5 primitives are: int, float, char, bool and string. String is simply a pointer that points to an contiguous chunk of characters terminated by null terminator.

3.2 Matrices

A matrix is a 2d array data container stored contiguously in memory. In order to make information such as the dimensions of the matrix easily accessible, each new matrix object also has an associated tuple. The tuple is contiguous in memory with and precedes the matrix object. The tuple stores information about height and width of the matrix. In fact, the tuple is really just a 1x2 matrix.
Matrices are initialized by declaration and populated using the square bracket notation. Users can index into a matrix as well as slice a matrix using the matrix operator. e.g

```
matrix M = [2,2;1,2]
M = [1;2,3] \\ Error
M[1,2] \\ indexing to value at column 1, row 2
M[0:2,1:2] \\ slicing a matrix along i:j,k:l
```

Matrices are mutable, but slicing them creates separate copies. Reassignment creates a shallow copy.

```
matrix C = M \\ Reassignment - makes shallow copy of M
matrix C2 = deep_copy(M) \\ Makes a deep copy of M
matrix C2 = M[0:M.width,0:M.height] \\ Another way to make a deep copy of M
```

4 Functions

4.1 User Defined Functions

Similar to C, YAMML supports user-defined functions. The user must specify the input and return types:

```
// math_operations.yamml
int add_one (int a)
    return a+1;
```

Functions return to caller by means of the return statement: namely

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

The user can declare any files they want to import with a # followed by the import keyword and the file name in angle brackets.

```
# import <math_operations.yamml>;
```

4.2 Function Calls

Functions are called by name and their argument in parentheses. User defined functions from other files need to be imported at the beginning of the file in order to be called.

```
\\ importing file with user defined function add_one() from above example
# import <math_operations.yamml>;


\\ calling user defined function with appropriate argument
add_one(1);
```

4.3 Variable Assignment from Functions

A function can be the right hand value of an assignment operator. In such a case, the left hand value of the assignment operator is assigned with the return value of the function on the right hand side.

```
# import <math_operations.yamml>; \\ importing file with user defined function add_one() from above

x = add_one(1); \\ x is assigned with the return value, 2, of function add_one()
```

5 Statements and Expressions

A YAMML program is made up of statements of the following types:

```
If-Else statement
While loop
For loop
Function Definitions
Expression
```

5.1 Statements

5.1.1 If...Else Statements

```
int a = 1;
int b = 3;
if (a <= b){
    print(a);
}
else{
    print(b);
}
```

5.1.2 While Statements

Identical definition with that of C's

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

Loop condition expression can be boolean or integer. Integer's other than 0 will be taken as true.

5.1.3 For Statements

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

5.2 Expressions

Expressions are parts of statements that are evaluated into expressions and variables using operators. These can be arithmetic expressions which includes an operand, an arithmetic operator, an operand, and a function call, which calls a function and returns a value.

5.3 Operators

5.3.1 Operator Types

YAMML suppports two types of operators,

5.3.2 Operator Precedence and Associativity

from lowest to highest precedence, the table below lists the operators and their associativities:

Operator	Meaning	Associativity
;	Sequencing	Left
=	Assignment	Right
||, &&	Or/And	Left
==, !=	Equality/Inequality	Left
<=, >=, <, >	Comparison	Left
+, −	Addition/Subtraction	Left
∗, /, .∗, ./	Multiplication/Division/Element-wise Matrix Multiplication/Division	Left
^	Exponentiation/Matrix Power	Right
$M[i, j]$	Matrix Indexing	Right
$M[i : j, k : l]$	Matrix Slicing	Right
!	Negation/Not	Right

6 Standard Library

6.1 Objects

YAMML has two types of built-in objects: matrices and strings.

6.1.1 Matrices

We include the following built-in functions for the matrix object to facilitate matrix algorithms.

```
returned_obj_type function_name (argument_type arg_name)
```

- int printm(matrix m): print the matrix
- matrix size(matrix m): returns [height, width]
- float sum(matrix m): sum of all the elements in the matrix
- float mean(matrix m): average of all the elements in the matrix
- matrix conv(matrix m, matrix k): convolve matrix with kernel
- matrix deep_copy(matrix m, matrix k): Deep copy matrix k into m.

- matrix pad(matrix m, int k): pad the input matrix with 0, return the new matrix.
- matrix find(matrix m): return a matrix of coordinates of non-zero values in matrix m.

6.2 Casting

The user can cast variables of one type to another. Because YAMML is strong-typed like C, all casts must be explicit. The syntax for type casting is similar to C:

```
var2 = (type) var1
```

The following casts are allowed:

```
(int) float -> int
(float) int -> float
(char) int -> char
(char) float -> char
```

7 Code Examples

```
// Applies a box blur to a matrix, perhaps an image
matrix boxblur (matrix input) {
    matrix kernel = 1/9 * [1,1,1;1,1,1;1,1,1];
    matrix result = conv(pad(input,1), kernel)
    return result
}

// A handy little algorithm or detecting boundaries for computer vision applications
matrix hough_accumulator (matrix edge_image, int theta_num_bins, int rho_num_bins) {
    matrix accumulator = zeros(rho_num_bins, theta_num_bins);
    int height;
    int width;

    matrix y;
    matrix x;
    [height, width] = size(edge_image)
    [y, x] = find(edge_image);
    max_rho = floor(sqrt(height^2+width^2)+1);
    del_rho = max_rho / (rho_num_bins/2);
    del_theta = pi/theta_num_bins;

    int cox;
    int coy;
    int rho;
    for (int i = 0; i < size(h)[0]; i+=1){
        cox = x[i];
        coy = y[i];
        for (int j = 0; j < theta_num_bin; j+=1){
            rho = coy*cos((j)*del_theta)-cox*sin((j)*del_theta);
            rho_bin_num = ceil(rho/del_rho + rho_num_bins/2);
            accumulator(rho_bin_num, j) = accumulator(rho_bin_num, j)+1;

        }
    }
    return accumulator
```

```
}
```

8 References

1. http://www.cs.columbia.edu/ sedwards/classes/2018/4115-fall/lrms/Coral.pdf