MPL: Matrix Processing Language

David Rincon-Cruz(Language Guru),
Chi Zhang(Tester),
Jiangfeng Wang(Team Leader),
Wode Ni(System Architect)
{dr2884, cz2440, jw3107, wn2155}

February 23, 2017

1 Lexical Elements

1.1 Identifiers

Identifiers are tokens used for naming varaibles and functions. They are case sensitive and should start with a letter and can follow with letters, digits or underscores. Below describes the definition of identifiers:

$$\begin{aligned} identifier &:= \texttt{(letter)(letter | digit | underscore)*} \\ digit &:= \texttt{'0'-'9'} \\ letter &:= \texttt{'A'-'Z''a'-'z'} \end{aligned}$$

1.2 Keywords

Keywords are case sensitive and are reserved for different uses in the language, so they cannot be used as identifiers. Below lists all the keywords in MPL:

int	float	boolean
true	false	print
#N	#S	#W
#E	#NW	#SW
#NE	#SE	#C
if	else	elseif
OR	NOT	AND
return	neg	Mat
Img	void	imgread
imgwrite	printm	new
while	func	null

1.3 Literals

Literals are constant string, numeric, or boolean values, such as "helloworld", 100, or false. Each literal has a specific type it belongs to and cannot be casted to other types. Assign a literal to another type that it does not belong to will cause an error.

1.3.1 Integer Literals

Integer literals are whole numbers represented represented by a sequence of 0-9 digits. An integer can be either positive or negative. To represent negative integers, a keyword "neg" is used.
Examples: 123; neg 321

1.3.2 Boolean Literals

Boolean literals have values either true or false. "true" and "false" are both reserved as keywords.
Examples: true; false

1.3.3 Operators

Operators are used for arithmetic operations such as addition, subtraction, multiplication and division. Operators can be applied on integers, float numbers and matrices.
We also add a few operators for easier manipulation of matrices calculation. `@` operator is used when applying a function to a matrix. `^`is used for calculating matrix transpose. `./` is used for element-by-element division in matrices. Similarly, `.*` is used for element-by-element multiplication.
The difference between `@` and `.@` is in regards to arguments. `.@` takes a function matrix and a value matrix and applies functions to corresponding entries in the value matrix. `@` takes a single function and applies it to every entry of a value matrix.
Examples: `+`; `-`; `*`; `/`;`@`;`.@`;`.*`

1.3.4 Delimiters

We use white space to separate different tokens in the code.

1.3.5 Parentheses and Braces

Parentheses and braces are used to better format the structure of code and limit the scope of variables. Local variables can only be accessed within the scope of code which is identified in the pair of curly braces.

1.3.6 Commas and Semicolons

Commas are used to separate function arguments. Semicolons are used to terminate a sequence of code.

1.3.7 Comments

Comment is denoted by `//`, such as `// COMMENT`
We also have block comment, which comments out section in between delimiters.

```
/* Comment
Yet another line of comment*/
```

2 Data Types

MPL uses strict typing. All variable types should be known at compile time and typecasting is not allowed.

2.1 Primitive Data Types

2.1.1 int

32 bit signed integer ranging from -2147483638 to 2147483647.

2.1.2 float

8-byte double-precision floating point numbers.

2.1.3 boolean

1-byte boolean type, either true or false.

2.2 Non-primitive Data Types

2.2.1 Img

Images are the next higher level structure to `Mat`. Images are a collection of "channel" matrices: R, G, B, A. Operations applied to Images are applied to each channel individually or each channel can be reference individually: `Img1.R`.

2.2.2 Mat

Matrices are the high level equivalent of their math counterparts and will be singly typed as one of 3: integer matrices, float matrices, and function matrices. Function matrices can be applied to integer and float matrices, and standard matrix operations apply (operators defined subsequently).

2.2.3 Declaring a matrix

You can declare a matrix by indicating the values at each entry with curly braces, and separate each row with a semicolon. All the entries in a matrix must be the same type, either int, float or boolean. For example, a 2*2 integer matrix can be initialized as:
`Mat int A = { 1 2; 3 4; }`

2.2.4 Accessing matrix entry

Matrix elements can be accessed by providing the row and column location within brackets next to the identifier of the matrix. Using the above example, to get the first row second entry (integer 2) in the matrix, we can do:
`A[0,1]`

3 Expressions and operators

3.1 Expressions

Expressions in MPL are made of operations between matrix and function. They are made up of one or more operands and operators. Like all other mathematics language, innermost expressions will be evaluated first. Otherwise the expressions with higher order will be evaluated before expressions with lower order. If expressions have the same order, the expressions will be evaluated from left to right.

3.2 Non-primitive Data Types

The tables below presents the language operators including assignment operators, mathematical operators, logical operators, comparison operators, logical operators. There are also descriptions and order:

operator	description	order
+	plus	1
-	minus	1
*	multiply	2
/	divide	2
=	assignment	0
<	Less than	0
>	More than	0
>=	Less than or equals	0
<=	More than or equals	0
==	equals	0
@	apply	3

4 Statements and Functions

4.1 Statement

4.1.1 The if Statement

The if statement is used to execute a statement if a specified condition is met. If the specified condition is not met, the statement is skipped over. The general form of an if statement is as follows:

```
if(condition) {
  statement1;
  statement2;
  statement3;
    // ...
}
else {
  statement1;
  statement2;
  statement3;
    // ...
}
```

4.1.2 The while Statement

The while statement is used to execute a block of code continuously in a loop until the specified condition is not maintained. If the condition is not met upon initially reaching the while loop, the code is never executed. The general structure of a while loop is as follows:

```
while(condition){
  statement1;
  statement2;
  statement3;
    // ...
}
```

4.2 Functions

4.2.1 User Defined Function Definitions

User Defined Functions in MPL is recognized as an operation on entries. It is treated more like type of data types. The user defined function will operate on and only on the entries. So the way it built is a little different from the tradition flows: a type key word which would be the type of the entry operated, an initial word of "func" but no return type, a function identifier and a paramter. However, in the func, it asks to return some data, which will be the resulted value of the entry operated. An example is shown below:

```
int func fblur {
  int temp = #C + 1;
  return temp ;
}
```

4.2.2 User Defined Function Calling

A user defined function can be used directly on a matrix, which will operate every entry in the matirx:

```
int func fblur {
  int temp = #C + 1;
  return temp ;
}
Mat int C = {1 1;
       1 1;}

Mat int Result = fblur @ C;

print Result;
// D will be {2 2;
//            2 2;}
```

4.2.3 User Defined Function Matrix

Functions can be used to build a function Matrix by giving functions in the entries of matrix instead of common data types like(int, double). Building user defined function matrix and use it on matrix is another way to calling functions. As mentioned before, function will operate only on entries. By using functions as the value of matrix, we can easily do different operations on different entries in the same matrix. An example is shown below:

```
int func f1 {
  int temp = #C + 1;
  return temp ;
}
int func f2 {
  int temp = #C + 5;
  return temp ;
}
int func f3 {
  int temp = #C - 7;
  return temp ;
}
int func f4 {
  return temp ;
}
// build a matrix .
Mat int A = { 1 2;
         3 4;}

// F is  a matrix of user defined functions .
Mat func F = { f1 f2;
           f3 f4 ;}

Mat int D = F @ C;

print D;
// D will be {1 5;
//           -6 1;}
```

4.2.4 System Function

The system function are some functions that are included in the language, which is the built-in functions and can be called in the main program. These are some useful and practical functions like "print", "scan","imgread". Usually they will not operate matrix entries. Details in 6.2 built-in functions.

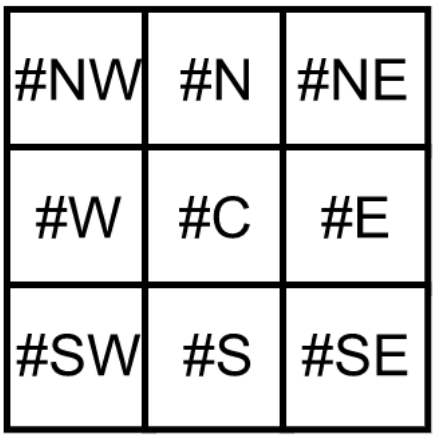

Figure 1: Layout of an entry and its immediate neighbors

5 Program structure and scope

5.1 Program structure

An MPL program is a sequence of function declarations and a sequence of statements, which are executed in order. All functions must be declared at the top of the program source file.

5.2 Scoping rules

All identifiers in MPL are in the global space, except the local variables declared inside functions.

Additionally, within the scope of an entry function, there are 9 predefined identifiers that represents the entry that the function is applied to and its immediate 8 neighbors: `#N`, `#S`, `#W`, `#E`, `#NW`, `#SW`, `#NE`, `#SE`, and `#C`. See Figure 1.

In the case of an 1×1 matrix and entries that are on the edges of a matrix, the neighbors that are outside of the boundry of the matrix will all have the special value `null` by default. It will be a runtime error to assign any value to these neighbors.

6 Built-in functions

6.1 `print` functions

```
print( string-literal );
printm( matrix );
```

The `print` function takes a string literal as input and will output it into `stdin`. `printm` function takes in a `Mat` typed value and prints the matrix in a human-readable format to `stdin`.

6.2 `imgread` and `imgwrite` functions

```
imgread( string-literal )
imgwrite( image, string-literal, string-literal )
```

`imgread` and `imgwrite` are I/O functions for images. `imgread` takes in a string literal containing a relative or absolute path to a file in the file system, opens the file, and returns an `Img` object containing the data of that file. `imgwrite` function takes in an `Img` object, a string literal specifying the file name, and another string literal specifying the file format of the image such as `"jpg"`. The function will then attempt to write the contents of the `Img` object to the file system using the specified format.

7 Sample programs

7.1 Image blurring

In this example, we construct a function that performs convolution on an image using a blur kernel encoded.

```
// Img is a basic struct that has 3 matrix.
// struct Img
// { Mat<double> R, G, B;}

// Read from a picture called Lenna.png
Img image = imgread("Lena.png");

// Now we have 3 matrix of int, R,G,B

// We enconde the convolution against blur kernel in this function
float func fblur {
  double grayValue=0.0;
  grayValue = #C*0.147761 + (#N+#S+#E+#W)*0.118318 + (#NE+#NW+#SE+#SW)*0.0947416
    return grayValue;
}

// Use the function fblur on every channel in the matrix;
Mat blurredR = fblur @ R;
Mat blurredG = fblur @ G;
Mat blurredB = fblur @ B;

// format the image for output
Img output = new Img(blurredR, blurredG, blurredB);

// imgwrite helps to output the image.
imgwrite(output,"blurredLena.png");
```

8 Context Free Grammar

The | and ∘ symbols are CFG syntax, not part of the language.

$$
\begin{aligned}
program &\rightarrow functionDefs \circ matrixCode \\
functionDefs &\rightarrow \epsilon | fDecls \circ functionDefs \\
fType &\rightarrow int | float \\
fDecl &\rightarrow fType \circ func \circ fId \circ \{gStatements\} \\
gStatements &\rightarrow \epsilon | gStatement; gStatements \\
fStatement &\rightarrow gExpr; \\
&\quad | return \\
&\quad | if(gExpr)\{fStatements\}else\{fStatements\} \\
&\quad | fvDecl \\
fvDecl &\rightarrow fType \circ id = fExpr; \\
fExpr &\rightarrow (fExpr) | fExpr + fTerm | fExpr - fTerm | fTerm \\
fTerm &\rightarrow fTerm * (fExpr) \\
&\quad | fTerm * number \\
&\quad | fTerm/(fExpr) \\
&\quad | fTerm/number \\
imgDecl &\rightarrow Imgid = Img(String); \\
&\quad | Imgid = Img(matId, matId, matId); \\
MatrixCode &\rightarrow genStatements \\
genStatements &\rightarrow \epsilon | matStatement | imgDecl | gExpr; \\
&\quad | if(gExpr)\{genStatments\}else\{genStatements\} \\
&\quad | while(gExpr)\{genStatements\} | return; \\
gvDecl &\rightarrow gType \circ id = gExpr; \\
gType &\rightarrow int | float | boolean \\
gExpr &\rightarrow (gExpr) | gExpr + gTerm | gExpr - gTerm | gTerm \\
gTerm &\rightarrow gTerm * (gExpr) \\
&\quad | gTerm * number \\
&\quad | gTerm/(gExpr) \\
&\quad | gTerm/number \\
matStatement &\rightarrow matDecl | fMatDecl | matExpr; \\
matDecl &\rightarrow Mat < type > id = matExpr; | \\
&\quad Mat < type > id = [matRows]; \\
matRows &\rightarrow [numbersList] | [numbersList]; matRows; \\
numbersList &\rightarrow number | number; numbersList \\
fMatDecl &\rightarrow fMatid = fMatExpr; \\
matExpr &\rightarrow (matExpr) \\
&\quad | matExpr + matExpr \\
&\quad | matExpr - matExpr \\
&\quad | matTerm \\
matTerm &\rightarrow matTerm * (matTerm) \\
&\quad | matTerm * matFuncted \\
&\quad | matTerm/(matExpr) \\
&\quad | matTerm/matFuncted \\
&\quad | matTerm * .(matTerm) \\
&\quad | matTerm * .matFuncted \\
&\quad | matTerm/.(matExpr) \\
&\quad | matTerm/.matFuncted \\
matFuncted &\rightarrow id \\
&\quad | fID@id \\
&\quad | fID@.id
\end{aligned}
$$