COMS W4115: Programming Languages and Translators

MATLIP: MATLAB-Like Language for Image Processing

Language Reference Manual

Pin-Chin Huang (ph2249@columbia.edu)
Shariar Zaber Kazi (szk2103@columbia.edu)
Shih-Hao Liao (sl2937@columbia.edu)
PoHsu Yeh (py2157@columbia.edu)

October 22, 2008

1. INTRODUCTION

Today, there are numerous image processing applications available such as Adobe Photoshop, Picture-it, Picassa, etc. However, these applications do not provide any programmatic ability to process images, which is required by image processing systems such as iris pattern recognition system. These image processing centric systems require conversion of image colors, image rotation, blurring, sharpening, resizing, edge detection and a lot of other processing. There are a number of languages available which offer such programmatic image processing, such as C++, Java, MATLAB, etc. Of these, only MATLAB provides ease of programming images for both novice and advanced users. However, because of the high cost in purchasing the license for MATLAB, it often deters many users in buying. Here, we propose a simple MATLAB like language syntax for simple and easy image processing. We call it MATLIP (***Mat**lab **L**ike **I**mage **P**rocessing*).

Since, our language will provide image manipulation, we start by a simple description of how images are represented in computers. A modern computer image, at the point where it is presented *(rendered)* for human consumption, usually consists of a rectangular array of closely spaced colored dots. Ideally, these dots are so small and so close together that the human eye cannot distinguish them individually. This causes them to run together and appear to represent continuous color. The individual dots are commonly referred to as pixels, which are derived from the term picture elements.

The pixels are typically stored and transported in files, and are then extracted from the files and displayed on a computer screen or a sheet of paper for human consumption. There are a fairly large number of formats for storing the pixels in a file. Different file formats have advantages and disadvantages in terms of compression, size, reproduction quality, etc. Our language will support reading standard image file formats such as JPEG, TIFF, BMP, GIF etc. by using our simple built-in function `imread()`. The language support provides manipulation of a single pixel, or a group of pixels, or an entire image. For the image we have a built-in type "`Image`". Since, often image processing requires a 2-D matrix which is applied over the image for various algorithms such as convolution, edge detection etc., we also have another built-in type "`Kernel`" to provide such functionalities.

2. TYPES

- **int:** The `int` data type is a 32-bit signed integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive).

- **double:** The `double` data type is a double-precision 64-bit floating point. It consists of an integer part, a decimal point, a fraction part, an `e`, and an optionally signed integer exponent. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the `e` and the exponent (not both) may be missing.

- **float:** The `float` data type is a single-precision 32-bit floating point. It consists of an integer part, a decimal point, a fraction part, an `e`, and an optionally signed integer exponent. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the `e` and the exponent (not both) may be missing.

- **boolean:** The `boolean` data type has only two possible values: `true` and `false`.

- **String:** The `String` data type represents a string of characters. It is internally implemented as an instance of the `String` class, but is exposed as a primitive data type in MATLIP. `Strings` are constant; their values cannot be changed after they are created.

- **Image:** The `Image` data type represents an image. It is internally implemented as an instance of the `BufferedImage` class, but is exposed as a primitive data type in MATLIP. Individual pixels are accessible using bracket annotation, `ID[x,y,"R"/"G"/"B"]`, to access a specific color channel of the pixel represented at the coordinate (x, y), or `ID[x,y]`, to access all three channels as a whole in the form of an integer.

- **Kernel:** The `Kernel` data type represents a kernel for image processing. It is internally implemented as an instance of the `Kernel` class, which is a matrix, but is exposed as a primitive data type in MATLIP.

3. LEXICAL CONVENTIONS

3.1 Reserved Keywords

The following is a list of reserved keywords in MATLIP. They may not be used for any other purpose.

`if`	`for`	`true`	`double`	`Kernel`
`elseif`	`while`	`false`	`float`	`String`
`else`	`break`	`function`	`boolean`	
`end`	`continue`	`int`	`Image`	

3.2 Identifiers

An identifier is a sequence of alphanumeric characters and underscores. Identifiers must start with a letter, and are case sensitive.

3.3 Special Characters

`[]`	Brackets are used to form kernels. For example, `[0.0, -1.0, 0.0; -1.0, 5.0, -1.0; 0.0, -1.0, 0.0]` defines a kernel for image sharpening. Moreover, brackets are also used to enclose subscripts of images.
`()`	Parentheses are used to indicate precedence in arithmetic expressions. They are also used to enclose arguments of functions.
`=`	Used in assignment statements.
`,`	Comma. Used to separate image subscripts, function arguments, and elements of a row in a kernel.
`.`	Decimal point. `314/100`, `3.14` and `.314e1` all represent the same value.
`:`	Colon. Used to create subscript arrays, and specify `for` iterations. For example, if `A` is a 100x100 image, one can "scissor-cut" a subset of the image and assign it to `B` using `B = A[25:50; 25:50];`. For-iterations can be

	specified by, for example, `for i=1:240.`
`;`	Semicolon. Used inside brackets to end rows for a kernel. Used after an expression to form a statement and to separate statements.
`%`	Percent. The percent symbol denotes a comment; it indicates a logical end of line. Any following text is ignored

4. EXPRESSIONS

An expression can be divided into five types, primary expression, unary expression, arithmetic expression, logical expression and relational expression.

4.1 Primary Expressions

Primary expressions can be an Identifier, Boolean value (`true`, `false`), an expression contained in parenthesis, an image, or an image element such as

```
Img[i, j, "R"]
```

which evaluates to an `int`.

4.2 Unary Expressions

A unary expression has the following form:

`unary_operator` *`expression`*

where `unary_operator` can only be the minus sign '`-`' in MATLIP, and *`expression`* can only be of type `int`, `float`, or `double`.

The unary expression composed of the operator, '`-`', followed by any other primary expression such as `boolean`, `Image`, or `Kernel`, is illegal.

4.3 Arithmetic Expressions

MATLIP has two main types of arithmetic operations. The first type applies to numerical values having the type of `int`, `float`, or `double`. The behavior of this type of operations is the same as that of C/C++ and Java. Take `a + b` for example. If both `a` and `b` are `integers`, the expression will also evaluate to an `integer`, If `integers` and `doubles` are mixed in an arithmetic expression, the result of the expression will be promoted to a `double`.

The order of precedence for this type of operations, from highest to lowest, is:

- `*` and `/`
- `+` and `-`

The second type of arithmetic operations may apply to images or kernels. Both operands can be images, or the expression can also be mixed with an image and a scalar value. The following summarizes the effects of this type of arithmetic operations, in a decreasing order of precedence:

`@`	▪ Convolution. It takes an `image` on the left and a `kernel` on the right, calculates a sum of products over an image using the kernel, and stores the results in a new image and returns it.
`*/`	▪ Multiplication/Division between two images; in this case, each pixel in the first image is multiplied/divided by the corresponding pixel in the second image, and the results are stored in the new image returned by the operation. ▪ Multiplication/Division between an image and a scalar value; in this case, individual pixel entries are multiplied/divided by the scalar value, and the results are stored in the new image returned by the operation. The scalar value can be an `int` or `float` or `double`, but it will always be implicitly converted to `int` by MATLIP.
`+-`	▪ Addition/Subtraction between two images; in this case, each pixel in the second image is added/subtracted to/from the corresponding pixel in the first image, and the results are stored in the new image returned by the operation. ▪ Addition/Subtraction between an image and a scalar value; in this case, the scalar value is added/subtracted to/from individual pixel entries, and the results are stored in the new image returned by the operation. The scalar value can be an `int` or `float` or `double`, but it will always be implicitly converted to `int` by MATLIP.

4.4 Relational Expressions

A relational expression has the following form:

`expression` `relational_operator` *`expression`*

where *`expression`* can be of type `int`, `float`, or `double`, and `relational_operator` can be one of the following, ‘`>`’, ‘`<`’, ‘`>=`’, ‘`<=`’, ‘`==`’, or ‘`~=`’, which represent “greater than”,” less than”, “equal or greater than”, “equal or greater than", "not equal" respectively in our language. Let A and B be two integer or double expressions. The logical expression, A>B, return true if A is greater than B else it will return false; The logical expression, A<B, return true if A is smaller than B else it will return false; The logical expression, A>=B, return true if A is equal or greater than B else it will return false; The logical expression, A<=B, return true if A is equal or smaller than B else it will return false; The logical expression, A==B, return true if A is equal to B else it will return false; The logical expression, A~=B, return true if A is not equal to B else it will return false.

The following summarizes relational operators:

>	▪ A>B returns true if A is greater than B else it will return false
<	▪ A<B returns true if A is smaller than B else it will return false
>=	▪ A>=B returns true if A is equal or greater than B else it will return false
<=	▪ A<=B returns true if A is equal or smaller than B else it will return false
==	▪ A==B returns true if A is equal to B else it will return false
~=	▪ A~=B returns true if A is not equal to B else it will return false

4.5 Logical Expressions

A logical expression has the following form:

```
expression logical_operator expression
```

where `expression` can only be of type `boolean`, and `logical_operator` can be one of the following, '&', '|', and '~', which represent "and", "or", "not" respectively in our language. Let A and B be two Boolean expressions. The logical expression, A&B, will return true if both A and B are evaluated to be `true`, else it will return `false`; The logical expression, A|B, will return `true` if A or B is evaluated to be `true`, else it will return `false`; The logical expression, ~A, will return `true` if A is evaluated to be `false`, else it will return `false`.

5. STATEMENTS

5.1 Expression Statements

A statement can be formed by terminating an expression with a semicolon ';'. Expression statements have the following form:

```
expression;
```

5.2 Group of Statements (Block)

A block is also a type of statement consisting of several statements. For example, a group of statements can be nested in the body of `if`-statement:

```
if expression
   statement+
end
```

5.3 Variable Declaration

Variable declaration has the following form:

```
TYPE identifier;
```

In MATLIP, a variable should always be declared first before it can be used. A variable without declaration should throw a syntax error. Also, a variable name cannot be declared twice. It is also illegal to declare multiple variables in the same line. For example,

```
int x;
Image A;
int x, y, z;  % This is illegal
```

5.4 Assignments

```
identifier = expression;
```

Our language only supports the assignment of an identifier of integer/double type. For example:

```
int x = 1;
int y = 1;
String z = ”R”;
Image A;
A[x, y, z] = 100;
```

The element of an Image is of integer type where x and y specify the coordinate of the pixel in the image, and z specifies the base color channel which we want to assign the value to. The color channel can only be either “R”, “G”, or “B”. In the example, we are assigning the “Red” channel of the pixel having the coordinate of (1, 1) to 100, and MATLIP will transform the new value to a single RGB `integer` type and then store it in the (1, 1) pixel.

The user can also assign a pixel value without specifying the color channel. In this case, the RGB value is assigned as a whole without transformation:

```
Image A;

% RGB value of pixel at (1, 1) is set to 100
A[1, 1] = 100;
```

The assignment between two images will copy the image on the right to the image on the left. Note that this is not just a reference copy; the image on the left will receive a new copy of the same image, and any subsequent changes to the new image will not affect the original. The following example copies one image to another:

```
Image A;
Image B;

A = imread(“lotus.jpg”);
% B receives a new copy of image A
B = A;
```

5.5 Conditional

A conditional statement can have the following form:

```
if expression
   statements
end
```

In this case, if the expression is evaluated to be true, then the program will execute the statements. Otherwise, the program will skip the statements and execute the line right after the end keyword.

Alternatively, a conditional statement can also have the following form:

```
if expression1
   statements1
elseif expression2
```

```
   statements2
else
   statements3
end
```

In the example, there is only one `elseif` statement. Generally, the user may use as many `elseif` statements as necessary following the `if`-statement and before the `else`-statement.

Using the example above, if `expression1` is evaluated to be true, `statements1` will be executed. If `expression1` is evaluated to be false while `expression2` is evaluated to be false, then `statements2` will be executed. If neither `expression1` nor `expression2` is true, then `statements3` will be executed. Note that if both `expression1` and `expression2` are evaluated to be true, then `statements1` will be executed while `statements2` will be skipped.

5.6 For Loops

A `for`-loop statement has the following form:

```
for variable = expression
   statements
end
```

In this case, `statements` in the body will be executed repeatedly while the `variable` is incremented each time, until the value of the `variable` falls out of the range of the expression. It is also possible to nest a `for`-loop inside another `for`-loop. For example,

```
for i=1:240
      for j=1:320
            zeros[i,j]=i;
      end
end
```

5.7 While Loops

A while-loop has the following form:

```
while expression
```

```
   statements
end
```

Here, statements in the body will be executed repeatedly until the `expression` is evaluated to be false.

5.8 Break

"`break;`" statement terminates the innermost loop; can only be used in for-loops and while-loops.

5.9 Continue

"`continue;`" statement skips all statements following it and returns to the beginning of the innermost loop; can only be used in for-loops and while-loops.

5.10 Function Calls

Function calls have the following form:

```
result = function_name(arguments);
```

or if there is no return value, simply

```
function_name(arguments);
```

where `result` is an identifier bound to a variable that is used to store the return value, `function_name` is an identifier bound to the function name, and `arguments` is a list of zero or more expressions separated by commas (`,`).

6. FUNCTION DEFINITIONS

Function definitions have the following form:

```
function return_value = function_name(argument_list)
    statements
end
```

where **`function`** and **`end`** are keywords, `function_name` is an identifier, *return_value* can have the form `(void | TYPE ID)` which describes the type of the return value and where the value is stored, and *argument_list* is a list of zero or more arguments having the form of `(TYPE ID)` separated by commas (`,`).

For example, the following function calculates the average of two doubles:

```
function double answer = avg(double x, double y)
    double sum = x + y;
    answer = sum/2;
end
```

Here, the function name is `avg`; `answer` is the return value; and `x` and `y` are the arguments taken by the function.

7. SCOPE & NAMES

7.1 Static Scoping

MATLIP uses static scoping. That is, the scope of a variable is a function of the program text and is unrelated to the runtime call stack. In MATLIP, the scope of a variable is the most immediately enclosing block, excluding any enclosed blocks where the variable has been re-declared.

7.2 Global vs. Local

Global variable: The variables declared outside of the function are global variables, which will be applied in the whole program except the function where there is a local variable with the same name as that of the global variable. Global variable will exist until the program terminates.

Local variable: The variables declared inside of the function are local variables, which will exist and be applied only inside that function.

Scope conflicts: If there is a global variable whose name is the same with that of the local variable, then the value of the local variable will be applied inside the function while the value of the global variable will be applied in all the other part of the program except that function.

For example:

```
int i = 0;  % global variable i with the value 0

function f(int i)
   i++;  % local variable with the value of 3 when f(2) is called
end; %end of scope of local variable i

f(2);
i = i + 1; % global variable i unchanged, now with a value 1
```

7.3 Forward Declarations

MATLIP requires forward declarations for variables and functions. That is, a variable needs to be declared before it can be referenced, and any function needs to be defined before it can be invoked.

For example, MATLIP generally prohibits the following and will throw an error:

```
double a;
double b;
double mean;

mean = func(a, b);

function double answer = func(double x, double y)
    ...
end
```

In this case, the function `func()` needs to be defined before it is called.

7.4 Arithmetic Operator Overloading

Arithmetic operators (`+, -, *, /`) are overloaded in MATLIP. They can be used in expressions where integers and floats/doubles are mixed, and where an image/kernel is mixed with a scalar value, as described in Section 4.3: Arithmetic Expressions.

The convolution operator (`@`), however, is not overloaded. It takes exactly an `image` on the left and a `kernel` on the right, nothing else.

7.5 Function Name Overloading

MATLIP does not allow function name overloading. That is, each function should have a unique function name, or MATLIP compiler will complain. This helps make a MATLIP program more readable and easier to understand, which are two important goals of the language.

7.6 Namespaces

MATLIP has only one namespace. Functions, variables, types, and record names, all share the same namespace. For example, a variable `foo` and a function `foo()` cannot coexist in a MATLIP program. This helps make a MATLIP program more readable and easier to understand, which are two important goals of the language.