iMatrix Reference Manual

Wenqi Jiang, Kaige Zhang, Zian Zhao

Contents

1 Lexical Elements

This chapter describes the lexical elements used in the language. They are called tokens and compiler will collect them to generate program, which can be analogous to picking words in a language to understand it. There are five types of tokens in iMatrix: identifiers, keywords, constants, operators, separators.[1]

1.1 Identifiers

Identifiers are the name you can set for the variable, function or your own structure. Identifiers should only contain letters, numbers and '_' underscore and have to start with a letter or underscore. Uppercase and lowercase letters are distinct so var and Var are two different variables.

1.2 Keywords

Keywords are special identifiers we reserved to make sure our language work properly. Now the keywords contain:

```
int float char string bool matrix image
if else for while break return struct
continue
```

1.3 Constants

A constant is numerical or character value.

1.3.1 Integer

An integer is a sequence of digits with the default base 10. For instance:

```
1 99 500
```

An integer constant will neglect the starting 0s, that is `0025` will be interpreted as `25`.

1.3.2 Real Number

A real number constant means a floating point number and it has to contain a '.' point in it. Followings are the valid real number constant examples:

```
float a;
a = 1.;
a = .15;
a = 1.2;
a = 1.e2;
a = -2.e-1;
```

1.3.3 Character

A character constant is a single character enclosed within single quotation marks, such as 'b'. For some special characters, such as quotation mark itself, you need add a back slash '\' before it as escape sequence. There are the special characters you need to take care of:

`\\`
: Backslash character.

`\'`
: Single quotation mark.

`\"`
: Double quotation mark.

`\n`
: Newline character.

`\t`
: Horizontal tab.

`\0`
: Termination character.

1.3.4 String

A string constant is a sequence of zero or more characters, digits, and escape sequences enclosed within double quotation marks. A string constant is of type "array of characters".

All string constants contain a null termination character ('\0') as their last character. Strings are stored as arrays of characters, with no inherent size attribute. The null termination character lets string-processing functions know where the string ends.

```
string a = "hello, world\n"
```

1.4 Operators

An operator is a special token that performs an operation, such as addition or subtraction, for more details you can go to Chapter 3.

1.5 Separators

A separator separates tokens. White space is a separator, but it is not a token. The other separators are all single-character tokens themselves:

```
( ) [ ]   ; ,
```

1.6 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. 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;
}
```

and

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

are functionally the same program.

Although you must use white space to separate many tokens, no white space is required between operators and operands, nor is it required between other separators and that which they separate.

```
/* All of these are valid. */

x++;
x ++ ;
x=y+z;
x = y + z ;
x=array[2];
x = array [ 2 ] ;
```

Furthermore, wherever one space is allowed, any amount of white space is allowed.

```
/* These two statements are functionally identical. */
x++;

x
      ++      ;
```

In string constants, spaces and tabs are not ignored; rather, they are part of the string. Therefore,

```
"potato knish"
```

is not the same as

```
"potato                    knish"
```

2 Data Types

2.1 Primitive Types

Primitive types are most basic types available in iMatrix.

2.1.1 Integer Types

- `bool` The 8-bit `bool` data type which is large enough to hold value 0 or 1.
- `char` The 8-bit `char` data type can hold integer values with the range of -128 to 127.
- `int` The 32-bit `int` data type can hold integer values with the range of -2,147,483,648 to 2,147,483,647.

2.1.2 Real Number Type

The `float` 64-bit data type represents fractional numbers. Its minimum value is stored in FLT_MIN and maximum value is stored in FLT_MAX.

The real number type provided here is of finite precision, and accordingly, not all real numbers can be represented exactly. Most computer systems use a binary representation for real numbers, which is unable to precisely represent numbers such as, for example, 4.2. For this reason, we recommend that you consider not comparing real numbers for exact equality with the == operator, but rather check that real numbers are within an acceptable tolerance.

2.2 Matrix

The `mat` data type represents a rectangular array of real number. In iMatrix, matrix elements are indexed beginning at position zero, not one.

2.2.1 Declaring Matrices

You declare a matrix by specifying its name, and its columns and rows enclosed with round brackets. Here is an example that declares a matrix.

```
mat a(3, 3);

mat b(1, 5);

mat c = b;
```

2.2.2 Initializing Matrices

In the declaration programmer can also specify the initial element value of a matrix, if `init` is not declared, default `init=0` will be applied. Programmer can also initialize the matrix

by directly assign every element to it, in this way you don't need to declare dimension. The elements in direct assignment should be wrapped up with brackets.

```
mat myMat(3, 3);
/*
 * [0., 0., 0.]
 * [0., 0., 0.]
 * [0., 0., 0.]
 */
mat myMat1(3, 3, init = 1);
/*
 * [1., 1., 1.]
 * [1., 1., 1.]
 * [1., 1., 1.]
 */
mat myMat2 = [[1, 2, 3],
              [4, 5, 6]];
/*
 * [1., 2., 3.]
 * [4., 5., 6.]
 */
```

2.2.3 Accessing Matrices Element

You can access the elements of a matrix by specifying the matrix name, followed by the element index, enclosed in brackets. We also provide slicing to access the part of an matrix. Here is an example:

```
myMat[0][1] = 5;

myMat[:][0];
```

This assigns 5. to the element of matrix in the first row and second column.

2.3 Image

The `img` data type is used to store any type of image files. It is an n-dimensional dense numerical single-channel or multi-channel array.[2]

We provide numerous handy operators and functions for these two data types and more detailed usages are in Chapter 3.

2.3.1 Declaring Images

Declaring a image is similar to declaring a matrix. You declare a image by specifying its name, and its height, width and channel enclosed with round brackets.

```
img image(50, 50, 3);
```

2.3.2 Initializing Images

By default, an image is initialized as all 0. You can also initialize it with another image either from a file or a pre-defined image.

```
/*An image with dimension 50*50*3 with initial value 0*/
img imageA(50, 50, 3);

/*Load image from local path*/
img imageB = image.load("/path/icon.jpg");

/*Initialize an image using pre-defined one*/
img imageC = imageB;
```

2.3.3 Accessing Images Element

You can access the elements of a image by specifying the image name, followed by the element index, enclosed in brackets. The number of pairs of brackets should be consistent with the number of dimensions. Additional, you can use slicing to access the part of an image. Here is an example:

```
img image(50, 50, 3);

/*Select the elements fron the first channel*/
image[:][:][0];

image[1:10][1:10][:];
```

2.4 Array

An array is a data structure that lets you store one or more elements consecutively in memory. In iMatrix, like most other languages, array elements are also indexed beginning at position zero, not one.

2.4.1 Declaring Arrays

You declare an array by specifying the data type for its elements, its name, and the number of elements (must be positive) it can store. Here is an example that declares an array that can store ten integers:

```
int myArray(10);
```

2.4.2 Initializing Arrays

You can initialize the elements in an array when you declare it by listing the initializing values, separated by commas, in a set of brackets. Here is an example:

```
int myArray(5) = [ 0, 1, 2, 3, 4 ];
```

You don't have to explicitly initialize all of the array elements. For example, this code initializes the first three elements as specified, and then initializes the last two elements to a default value of zero:

```
int myArray(5) = [ 0, 1, 2 ];
```

If you initialize every element of an array, then you do not have to specify its size; its size is determined by the number of elements you initialize. Here is an example:

```
int myArray = [ 0, 1, 2, 3, 4 ];
```

2.4.3 Accessing Arrays Element

You can access the elements of an array by specifying the array name, followed by the element index, enclosed in brackets. Remember that the array elements are numbered starting with zero. You can use slicing to access a sequence of elements in array as well. Here is an example:

```
myArray[0] = 5;

myArray[:4];
```

2.5 Structure

A structure is a programmer-defined data type made up of variables of other data types (possibly including other structure types).

2.6 Defining Structures

You define a structure using the `struct` keyword followed by the declarations of the structure's members, enclosed in braces. You declare each member of a structure just as you would normally declare a variable—using the data type followed by one or more variable names separated by commas, and ending with a semicolon. Then end the structure definition with a semicolon after the closing brace.

You should always include a name for the structure in between the `struct` keyword and the opening brace.

Here is an example of defining a simple structure for holding the X and Y coordinates of a point:

```
struct point
{
    int x, y;
};
```

That defines a structure type named `struct point`, which contains two members, x and y, both of which are of type int.

Structures may contain instances of other structures, but of course not themselves.

2.7 Declaring Structures

You can only declare variables of a structure type after defining the structure by using the `struct` keyword and the name you gave the structure type, followed by one or more variable names separated by commas.

```
struct point
{
    int x, y;
};
struct point first_point, second_point;
```

That example declares two variables of type struct point, first_point and second_point.

2.8 Initializing Structures Members

You can initialize the members of a structure type to have certain values when you declare structure variables.

One way to initialize the members is to specify the name of the member to initialize. This way, you can initialize the members in any order you like, and even leave some of them uninitialized.

```
struct point first_point = { .y = 10, .x = 5 };
```

2.9 Accessing Structures Members

You can access the members of a structure variable using the member access operator. You put the name of the structure variable on the left side of the operator, and the name of the member on the right side.

```
struct point
{
    int x, y;
};

struct point first_point;

first_point.x = 0;
first_point.y = 5;
```

You can also access the members of a structure variable which is itself a member of a structure variable.

```
struct rectangle
{
    struct point top_left, bottom_right;
};

struct rectangle my_rectangle;

my_rectangle.top_left.x = 0;
my_rectangle.top_left.y = 5;

my_rectangle.bottom_right.x = 10;
my_rectangle.bottom_right.y = 0;
```

3 Expressions and Operators

3.1 Logical operators

Operators for bool type

Operators	Description
&&	logical AND
||	logical OR
!	logical NOT

For example:

```
(a > 0) && (b > 0);

isCaseA || isCaseB;

!(a = false);
```

3.2 Arithmetic Operators

Operators for int and float type

For example:

```
a++;

++a;

/* = 7 */
1 + 2 * 3;
```

Operators	Description
+	Addition
-	Subtraction
++	Add 1
- -	Subtraction 1
*	Multiplication
/	Division
%	Modulo
^	Power

```
/* = 2*2*2 = 8 */
2^3;
```

3.3 Matrix Operators

Operators for Matrix type

Operator "+', "-", "*", "/" can be used as:
matrix operator matrix (element wise)
matrix operator value
value operator matrix
Matrix-Matrix Multiplication is for 2-dimensional matrix only. Operations between matrices with inconsistent dimensions will be simply output error.

Operators	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
*.	Matrix Multiplication
[ ]	Index

For example:

```
/* matA is M col N row, matB is M col N row */
/* matC[i][j] = matA[i][j] + matB[i][j] */
matC = matA + matB;

/* matB[i][j] = a * matA[i][j] */
matB = a * matA;
matB = matA * a;

/* matA is M col N row, matB is M col N row */
```

```
/* matC[i][j] = matA[i][j] * matB[i][j] */
matC = matA * matB;

/* matA is M col N row, matB is N col P row */
/* matC[i][j] = matA[i][0] * matB[0][j] + ... + matA[i][N] *
    matB[N][j] */
matC = matA *. matB;

/* Index & Slicing */
val = matA[0][0];
matB = matA[0:2][0:2];
```

3.4 Image Operators

Image Operators is a subset of Matrix Operators.
Similar to the operators of matrix, the operations only between consistent images will be handled correctly. The number of index brackets should always be consistent with the actual dimension of an image.

Operators	Description
+	Addition
-	Subtraction
[ ]	Index

3.5 Relational Operators

Operators	Description
>	Greater Than
<	Smaller Than
==	Equal To
<=	Greater Than Or Equal To
>=	Smaller Than Or Equal To
!=	Not Equal

4 Statements

4.1 if-else

```
if (condition)
    {then_statement}
else
    {else_statement}
```

For example:

```
if (a<0)
    {a = -a;}
else
    {a = a;}
```

4.2 while Loop

```
while (condition)
    {loop_statement}
```

For example:

```
while (a<10)
    {a = a+1;}
```

4.3 for Loop

```
for (init; condition; step)
    {loop_statement}
```

For example:

```
for (int i = 0; i<10; i++)
    {a = i;}
```

4.4 break

For example:

```
while (1)
    {
        i++;
        if (i>10) break;
    }
```

4.5 continue

For example:

```
for (int i = 0; i<N; i++)
    {
        if (a[i]<0) continue;
        sum = sum + a[i];
    }
```

5 Functions

This chapter introduces the rules of functions: how to define and declare; what parameters to pass in; how to call a function; anonymous functions, etc.

5.1 Function Declarations

A function declaration is used to specify the name of a function, a list of parameters, and the function's return type. A sample of declaration is like below.

```
float partialSum(int times, float val);
```

In the function declaration above, the first part 'float' is the return type. Return type can be any primitive type or user-defined type. If the function has nothing to return, use the keyword 'void' as return type.

The 'partialSum' is the function name, which can be expressed by any identifier.

The parameter list is 'int times, float val', which is scoped by a pair of parenthesis. This list can contain any number of parameters include zero. Each parameter should be declared with it's type.

Finally, the declaration ends up with a semi-colon. If the declaration comes up with it's definition, then we don't write a semi-colon anymore, as in 5.2.

5.2 Function Definitions

Function definitions specifies what a function does. It not only contains the name, parameter list and return type of a function, it also includes the function body.

```
float partialSum(int times, float val) {

int thisStep = times * times;
float result = val + thisStep;

return result;
}
```

In the example above, the function body clarifies how the function works.

5.3 Calling Functions

A function can be called by a format like

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

Also, we can feed the return value of a function to a variable or discard the result.

```
float result = partialSum(10, 5.3); /* assign to a variabl */
partialSum(10, 5.3);                /* discard result */
```

5.4 Function Parameters

Function parameters can be primitive types and user-defined types. Functions can not be passed into another function as parameters.

These parameters are local copy within the function body, not the references to original variables, i.e. reassign values to these parameters in the function body will not influence the orginal variabls.

```
void changeParams(int a, float b) {
    a = a + 1;
    b = b - 1;
}

int main () {
    int a = 1;
    int b = 1.2;

    changeParams(a, b);

    print("%d \n", a); /* still 1 */
    print("%f \n", b); /* still 1.2 */
}
```

5.5 Variable Length Parameter Lists

iMatrix allows users to set some default value to part or all of the function parameters. These default parameters can only be primitive types, not user-defined types. This enables user to neglect some trivial arguments they don't want bother to pass in. For example, there are tons of parameters in a edge detection algorithm. A user can call the function without knowing every details about these hyper-parameters.

```
img imgProcess(img image, int threshold=50) {
    /* initialize a new image with same size, set default value to 0 */
    img newImage(image.height(), image.width(), 0);
    for (int i = 0; i < image.height(); i++) {
        for (int j = 0; j < image.width(); j++) {
            if (image[i][j] >= threshold) {
                newImage[i][j] = image[i][j];
            }
        }
    }
}
```

When calling this function, we can either fill in this parameters or not.

```
imgProcess(myImg);    /* not pass threshold value, use default */
imgProcess(myImg, 100); /* pass threshold into the function */
```

5.6 The 'main' Function

Each program should contain a function named 'main'. The main function does not necessarily need to have a concrete function body, but we need to define it.

The return type for main is always int. A return value of 0 tells the system the program has been executed successfully. Other return values, e.g. -1, report errors to the operating system.

```
int main() {
    /* a main function without function body */
    return 0;
}

int main() {
    /* with function body & error detection */
    img a(1080, 768, 0);
    img b = imgProcess(a, 80);

    if (errorDetection() == true)
        return -1;
    else
        return 0;
}
```

If user does not declare the return value in main function, the main function will return 0 by default.

```
int main() {
    /* with function body & error detection */
    img a(1080, 768, 0);
    img b = imgProcess(a, 80);

    if (errorDetection() == true)
        return -1;

    /* by default, main function will return
     * a 0 to the operating system */
}
```

5.7 Recursive Functions

iMatrix allows users to define recursive functions, i.e. call the function itself within it's function body.

```
int addUp(int n) {
    if (n == 1)
        return 1;
    else
```

```
        /* call itself within the function body */
        return addUp(n - 1) + 1;
}
```

5.8 Anonymous Functions

We can define anonymous function to help user write cleaner code. Anonymous function can be especially useful when doing simple operations.

```
/* lambda keyword, return type, parameters, function body, arguments */
int a = (lambda int (int a) { return a + 1;}, (10));

img afterprocess =
    /* lambda declaration */
    (lambda img (img image, int delta) {
    /* function definition */
        for (int i = 0; i < image.height(); i ++)
            for (int j = 0; j < image.width(); j++)
                image[i][j] += y;
        return image; }
    /* arguments passed in */
    , (myImg, 10));

/* a more clean version,
 * which takes advantage of operator overload */
img afterprocess2 =
    (lambda img (img image, int delta) {
        return delta + image; }
    , (myImg, 10));
```

6 Program Structures and Scope

6.1 Program Structure

A iMatrix program may contain everything within a same file.But more commonly, large programs consist of several header files and source files. Header files should contain the function declarations corresponding to it's source file, while source files provides full definitions of these functions. If you don't want some functions to be called by other files, you may not declare them in header files.

```
/* a typical structure of a iMatrix file */
#include "imageProcessing.h"

img process1(img image) {
    /* do something here */
}
```

```
img process2(img image) {
    /* do something here */
}
```

6.2 Scope

A variable is visible only within a particular function. We currently do not allow global variables and static variables. Variable must be passed through parameters between functions.

```
int function(int a) {
    int b = 1;
    return a + b;
}

int main() {
    int b = 5;
    /* b will not be passed in function b */
    function(10);
    print("%d \n", b); /* b is still 5 */
}
```

7 Sample Programs

7.1 Hello World

```
int main()
{
    string message = "hello world";
    print("%s", message);
}
```

7.2 Matrix Solution

```
/* solve AX = B */
mat A = [[1, 2, 3], [4, 5, 6]];
mat B = [[3, 6]];
mat X = Inverse(A) *. B;
```

7.3 Image Preprocessing

Before implementing algorithms on an image, we usually preprocess the image for a better result. For example, some images are seriously dominated by noise. In this case we may need some preprocessing algorithm to get rid of these annoying noises.

```
img input = image.load("sample_img.jpg");
```

```
img output = image.filter(input,
                  class="average", kernelSize=3);
image.save("adjusted_img.jpg", output);
```

7.4 Edge Detection

Edge detection is a commonly used function in image processing. We provide a sample code of a simple algorithm for edge detection. The functions in the sample code below are all built-in functions.

```
img input = image.load("sample_img.jpg");
img grad = image.grad(input, sampleRange=1);
/* return a matrix only contain 0 and 1,
 * the selected points will be label as 1 */
img index = image.selectByThreshold(grad,
                  larger=true, threshold=50);
/* here we define the gradients larger than
 * 50 can be selected */
img output = input * index;
image.save("edge.png", output);
```

7.5 Histogram

Compute the histogram of images is a usual and useful function in image processing. These histograms can be used for edge detection, image segmentation, etc. Thus, we will develop a built-in function for histogram computation.

```
img input = image.load("sample_img.jpg");
mat hist = image.histogram(input);
print("%m\n", hist);
```

8 References

References

[1] The GNU C Reference Manual. Retrieved from https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

[2] OpenCV 2.4.13.7 documentation. Retrieved from https://docs.opencv.org/2.4/index.html