IML Language Reference Manual

Steven Chaitoff Zach van Schouwen Cindy Liao Eric Hu

A. Introductory Notes

IML is an image manipulation programming language. More exactly, IML is a Turing-Complete language that can open image files, manipulate the pixel contents, and save them. Alongside mathematical and logical operators, operators exist for manipulating color and transparency bit values in pixels. IML supports image editing only at the pixel level, but it has flow control instructions such as conditionals, iteration and recursion, and programmers may define their own functions. Given these constructs, it is possible to create programs for a variety of image modifications such as cropping, resizing, shearing, perspective transformations, and other oft-found operations in graphical editing programs. Automated "batch" processing is a particularly suitable use of IML.

IML as an interpreted command-line language. It works with images as raw bitmap data, and the interpreter acts as an interface between raw blocks of pixels and formatted input and output files.

B. Lexical Description

B1. Tokens

There are six types of tokens: identifiers, keywords, constants, string literals, operators, and separators. Comments are ignored. Any white space (defined as any number of spaces, tabs, newlines or feeds) is also ignored, except as to separate adjacent tokens where no white space separation would be ambiguous.

B2. Comments

There are two types of comments. Single-line comments begin with `//` characters and continue until a newline character. Multi-line comments may begin and end with `/*` and `*/`, respectively. Comments do not nest.

B3. Identifiers

Identifiers are any sequence of one or more letters, numbers, and underscores. The first character of an identifier must not be a number. Upper and lower case letters are different. Identifiers may be any length.

B4. Keywords

The following words are reserved and may not be used as identifiers.

```
alpha    blue   break   cols     else
green    for    if      length   red
return   rows   save    void     while
```

B5. Constants

B5.1. Integer Constants

Integer constants consist of a sequence of one or more digits optionally preceded by a `-` character, assumed to be in base 10.

B5.2. Floating Point Constants

Floating points constants consist of an integer part, a decimal point, a fraction part, an `E` character, and exponent part. The fraction and exponent parts consist of a sequence of one or more digits. The integer part consists of a sequence of one or more digits optionally preceded by a `-` character. Either the integer part or fraction part may be missing, but not both. Either the decimal part or the `E` and exponent may be missing, but not both.

B6. String Literals

String literals are any sequence of characters beginning and ending with a `"` character. Any `"` characters that are not at the beginning or end of the sequence must be preceded by a `\` character.

B7. Separators

Separators are tokens that serve to delimit the position of other tokens.

B7.1. Delineators

`(` and `)` characters surround expressions and argument lists for function calls. `{` and `}` characters delineate the scope of identifiers inside of them. The `,` character separates both arguments in a function call and parameters in the argument list of a function declaration.

B8.2. Terminators

Every statement that does not precede a statement-block must end with the `;` character.

C. Syntax Notation

C1. Basic Types

C1.1. Pixel Type

The Pixel type stores one pixel in memory, which consists of red, green, blue and alpha (RGBA) integer values. Red, green, and blue intensity values range from 0 to 255. The alpha transparency value ranges from 0 to 100. Any RGBA value set to greater than maximum value will revert to its maximum value, and any value set to less than 0 will revert to 0. The default RGBA value of a Pixel is 255 red, 255 green, 255 blue, and 100 alpha, which is a white and opaque pixel. A pixel's values can be accessed via the channel operators described in section D5.

C1.1. Image Type

The Image type stores an image in memory, which consists of a two-dimensional array of Pixel types.

The default value of an Image is a 1x1 array composed of one Pixel. The size of an image can be accessed via the rows and cols operators, described in sections D4.2 and D4.3.

C1.1. Integral Type

Integral type (Int) stores a 32 bit signed integer in memory. Values can range from -2147483648 to +2147483647. The default value of an integer is 0.

C1.1. Floating Point Type

The Floating Point type (Float) stores a 64 bit double precision floating point number in memory. The default value for a floating point is 0.0.

C1.1. String Type

The String type is used to store character strings in memory. The default value of a string is the empty string, "". A string can be initialized by setting it to a literal:

```
String hello;  hello = "hello world";
```

C2. Derived Types

C1.2. Array Types

It is possible to create an array of any type, including an array itself (multi-dimensional arrays). Array types are denoted with square brackets ([ ]) following an identifier. The number of elements in an array is static; this number must be specified during declaration, and it cannot be changed during execution.

The size of an array is accessed with the length operator described in D4.1. Array indices begin at 0. Individual elements are accessed by the array reference operator, described in D2.1. No syntax errors are generated for attempting to access elements outside the bounds of an array.

C1.2. Function Types

Functions provide a convenient way to encapsulate some computation without worrying about its implementation. Functions are composed of a optional sequence of input arguments, a series of statements, possibly including other function calls and recursive calls, and at most one return value. All functions must defined before they can be called.

C3. LValues

An *lvalue* is an expression referring to an *object*, or a named region of storage. All basic types and array types yield an lvalue equal to its stored value. This allows assignments like the following:

```
Int a;    Int b;    a = b = 5; // b is assigned 5 and also returns a value of 5
```

The lvalue of a function is specified during the function definition (see section E2.2).

C4. Conversions

Only native conversions as described below are supported. Any illogical conversions return a syntax error. Explicit conversions return a syntax error as well.

C4.1. Casting

No explicit casting is allowed.

C4.2. Arithmetic Conversions

Integers, when assigned to floating point types, will convert to a floating point type. Floating point types, when assigned to integers, will have any digits following the decimal point truncated and then converted to an integer.

C4.3. Integral and String Conversions

Integers, when assigned to strings, will be converted to a string containing the value of the integer. Strings cannot be assigned to integers and doing so will result in a syntax error.

C4.4. Pixel and Image Conversions

Assigning a pixel to an image will result in the pixel being promoted to an Image type of size 1x1 containing that pixel. Images cannot be demoted to a pixel.

D. Expressions

The operators defined in the following expressions appear in decreasing precedence.

D1. Primary Expressions

A primary expression is any identifier, constant or parenthesized expression, which may itself be a primary expression. Therefore, a primary expression consists of any sequence of identifiers, constants and operator expressions ordered by the associativity and precedence rules of the operators therein.

D2. Postfix Operators

D2.1. Array References

Array references are indicated by square brackets ([ ]) following the array identifier on which they operate. They are in postfix form and are left-to-right associative, meaning multi-dimensional references are expanded outwardly:

```
( a[0][1][2] == (((a[0]) [1]) [2]) ) // evaluates true: a is a three dimensional array of size >= 1x2x3
```

Array references on Image types allow access to Pixel types:

```
Image myimage  Pixel mypixel;  mypixel = myimage[0][0]; // upper left most pixel of image
```

Array references on String types return a string containing the character referenced:

```
String h; h = hello[0]; // h has value "h"
```

D2.2. Function Calls

Function calls are postfix expressions consisting of a comma-separated list of argument expressions:

```
Int a;    Int b;
a = 1;    b = 1;
sum (1, 1) ;   // function call
```

D3. Unary Operators

D3.1. Unary Minus Operator

The unary minus (-) is a prefix operator that negates integer and floating point values. It has right-to-left associativity.

D4. Sizing Operators

Sizing operators are prefix operators and have right to left associativity. Sizing operators do not return lvalues, as arrays are fixed in size.

D4.1. Length Operator

The length operator can only be applied to array types. It returns the number of elements in an array. Although the Image type behaves like a two-dimensional array and the array reference operators can be applied to it, the length operator cannot because it is not an array type.

```
Image a[10];   (length a == 10)    // evaluates true: a is an array of 10 Image types
```

D4.2. Rows Operator

The rows operator can only be applied to Image types. It returns the height of the image in pixels.

D4.3. Cols Operator

The cols operator can only be applied to Image types. It returns the width of the image in pixels.

D5. Channel Operators

Channel operators are prefix operators and have right to left associativity. Channel operators can only be applied to Pixel types, and they return lvalues, therefore channel properties of a Pixel can be modified.

D5.1. Red Channel Operator

The red operator returns the red intensity value of a Pixel:

```
Pixel a;  red a = 0;     // remove the red hue from a pixel
```

D5.2. Green Channel Operator

The red operator returns the green intensity value of a Pixel.

D5.3. Blue Channel Operator

The red operator returns the blue intensity value of a Pixel.

D5.4. Alpha Channel Operator

The red operator returns the alpha transparency value of a Pixel.

D6. Multiplicative Operators

Multiplicative operators are multiplication (+), division (/) and modulus (%). They are infix operators with left-to-right associativity. Multiplication demands arithmetic operands. If both operands are Int types, the result is an Int type. If either operand is a Float type, the result is a float type. Division also demands arithmetic operands. The result is always a Float type. Modulus demands Int type operands, and the result is always an Int type.

D7. Additive Operators

Additive operators are addition (+) and subtraction (-). They are infix operators with left-to-right associativity. In addition of arithmetic operands, the result is the sum of boh operands. If both operands are Int types, the result is an Int type. If either operand is a Float type, the result is a float type. In addition of String operands, the result is the concatenation of the right-hand String to the left-hand String. In addition of an Int operand to a String operand (String + Int), the Int operand is converted to a string as described in section C4.3, and the two Strings are concatenated.

The subtraction operator demands arithmetic operands. If both operands are Int types, the result is an Int type. If either operand is a Float type, the result is a float type

D8. Relational Operators

D8.1. Comparison Operators

Comparison operators are less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=). Comparison operators are infix operators and have left-to-right associativity. They demand arithmetic operands. They return the value 1 if the relationship between their operands is in fact true, and 0 if it is false.

D8.2. Equality Operators

Equality operators are equal to (==), and unequal to (!=). They behave in the same way as comparison operators except with lower precedence.

D9. Logical Operators

Logical operators are infix operators and have left-to-right associativity.

D9.1. And Operator

The logical and operator (&&) returns false if any of its operands are false. Otherwise it returns true. False operands consist of 0 (Int), 0.0 (Float), and "" (an empty String). True operands are consist of all non-false operands.

D9.2. Or Operator

The logical or operator (||) returns false if all of its operands are false. Otherwise it returns true.

D10. Assignment Operator

The assignment operator (=) is an infix operator with right-to-left associativity. Its left-hand operand must be an lvalue, as described in section C3. Assignment puts the value of the right-hand operand into the storage space pointed at by the left-hand operand.

D11. Save Operator

The save operator is an infix operator with right to left associativity. It saves an Image to disk and must only be applied in the form (Image save String) where String indicates the filepath of the image file:

```
Image a = "/path/to/photo.jpg";
a save "/path/to/copy.jpg";
```

E. Declarations

E1. Type-specifiers

Variables are defined by prepending a type specifier to the name of the new variable:

type-specifier identifier ;

All variables must be forward-declared; doing otherwise will result in a runtime error.

E2. Declarators

E2.1. Array Declarators

Arrays are declared in a similar manner to basic types, but a length specification is required.

type-specifier identifier **[** *expression* **]** *;*

E2.2. Function Declarators

Functions are defined in the following manner:

type-specifier postfix-expression **(** *argument-expression-list?* **)**
statement-block

The argument expression list takes the form of a comma-delimited sequence of variable declarations:

parameter-list := *type-specifier identifier* **[,** *type-specifier identifier* **] ***

Function names are subject to the same language constraints as variable names, as they are identifiers.

E3. Initialization

Initialization for variables must not be carried out at declaration time, and it must be handled in a separate statement:

```
Int n;     // initialization
n = 10;    // assignment
```

E3.1. Image Initialization

An Image is initialized by assigning to it a string literal of the path of a supported image file:

```
Image myimage; myimage = "/path/to/photo.jpg";
```

A path to a non-image file or an unopenable image file is a runtime error.

F. Statements

Statements are generally expressions followed by a semicolon to represent the end of a statement. Statements are executed in sequence. Statement-blocks are sequences of zero or more statements.

F1. Selection Statements

Selection statements evaluate conditions and direct control flow appropriately.
Valid selection statement forms are:

> **if (** *expression)* *statement-block*
> **if (** *expression)* *statement-block* **else** *statement-block*

F2. Iteration Statements

Basic iteration structures are supported for looping control flow.

F2.1. For Loops

A valid for statement form is:

> **for (** *expression-statement expression-statement expression-statement* **)** *statement-block*

The first statement is evaluated before the loop begins; the expression is evaluated at the beginning of each iteration (and, if false, ends loop execution); finally, the last statement is evaluated at the end of each iteration.

F2.2. While Loops

A valid while statement form is:

> **while (** *expression* **)** *statement-block*

At the beginning of each iteration, if the qualifying expression evaluates to true, statement-block is executed.

F2.3. Break

The break keyword must only be placed in the statement-block of an iteration statement. When encountered, execution immediately breaks out of the iterative loop as if the qualifying expression had returned false. The current loop of iteration is not completed. That is, the statements following the break statement in that block are not executed.

F3. Compound Statements

Nested statements are permitted, such that selection and iteration statements can appear inside of a statement block. All statement blocks must begin with an open bracket and end with a close bracket.

F3.1. Identifier Scope

The scope of a variable is determined by the deepest statement block in which it appears. An identifier is bound (defined) between the open bracket above it that is closest to it and the corresponding closed bracket.

G. Grammar

Nonterminal symbols are written in *italics*. Terminals are written in **boldface**.
A question mark (*?*) character following a non-terminal indicates that it is optional.

statement:
> *statement-block*
> *selection-statement*
> *iteration-statement*
> *expression-statement*
> *declaration-statement*

statement-block:
> **{** *compound-statement* **}**

compound-statement:
> *statement*
> *compound-statement statement*

selection-statement:
> **if (** *expression* **)** *statement-block*
> **if (** *expression* **)** *statement-block* **else** *statement-block*

iteration-statement:
> **while (** *expression* **)** *statement-block*
> **for (** *expression-statement expression-statement expression-statement* **)** *statement-block*

declaration-statement:
> *type-specifier identifier size-specifier?* **;**
> *function-type-specifier identifier (argument-expression-list?) statement-block*

function-type-specifier:
    void
    type-specifier

type-specifier:
    Int | **Float** | **String** | **Pixel** | **Image**

size-specifier:
    [*expression* **]** *size-specifier*
    [*expression* **]**

expression-statement:
    expression? **;**

expression:
    assignment-expression
    unary-expression **save** *unary-expression*

assignment-expression:
    logical-or-expression
    unary-expression **=** *logical-or-expression*

logical-or-expression:
    logical-and-expression
    logical-or-expression **||** *logical-and-expression*

logical-and-expression:
    equality-expression
    logical-and-expression **&&** *equality-expression*

equality-expression
    relational-expression
    equality-expression **==** *relational-expression*
    equality-expression **!=** *relational-expression*

relational-expression:
    additive-expression
    relational-expression **<** *additive-expression*
    relational-expression **>** *additive-expression*
    relational-expression **>=** *additive-expression*
    relational-expression **<=** *additive-expression*

additive-expression:
    multiplicative-expression
    additive-expression **+** *multiplicative-expression*
    additive-expression **-** *multiplicative-expression*

multiplicative-expression:
    unary-expression
    multiplicative-expression ***** *unary-expression*
    multiplicative-expression **/** *unary-expression*
    multiplicative-expression **%** *unary-expression*

unary-expression:
    channel-expression
    - *unary-expression*
    length *channel-expression*
    rows *channel-expression*
    cols *channel-expression*

channel-expression:
    postfix-expression
    channel-operator postfix-expression

postfix-expression
    primary-expression
    postfix-expression **[** *expression* **]**
    postfix-expression **(** *argument-expression-list?* **)**

primary-expression
    identifier
    constant
    (*expression* **)**

argument-expression-list:
    assignment-expression
    argument-expression-list **,** *assignment-expression*

constant:

integer-constant
string-constant
floating-constant

H. Sample Code

H1. Converting an Image to Grayscale

```
// Takes an Image as input, converts its pixels to grayscale and returns it.
// Iterates through the image's pixels in nested for loops, and for each pixel
// calculates an average color intensity and sets RGB channels to that intensity
//
Image grayscale (Image input)
{
     Int i ;
     Int j ;
     ( for i = 0; i < rows input; i = i + 1 )
     {
          ( for j = 0; j < cols input; j = j + 1 )
          {
               Float avg ;
               Pixel pxl;
               pxl = input[i][j] ;
               avg = ( red pxl + green pxl + blue pxl ) / 3.0 ;
               red input[i][j] = green input[i][j] = blue input[i][j] = avg ;
          }
     }
     return input ;
}
```

H2. Batch Processing Several Images

```
// Takes String path and Int n as input, that point
// to n images at location path whose filenames are
// labeled image1.jpg, image2.jpg...imagen.jpg.
// Edits and saves the edited images to that folder.
//
void batch (String path, Int n)
{
     Int i;
     ( for i = 1; i <= n; i = i + 1 )
     {
          Image temp ;
          temp = path + "/image" + n + ".jpg" ;
          grayscale ( temp ) save path + "/image" + n + "_gray.jpg" ;
     }
}
```