The pixelman Language Reference Manual

Anthony Chan, Teresa Choe, Gabriel Kramer-Garcia, Brian Tsau

October 2017

Contents

1 Introduction

pixelman is a language used mainly for the purpose of image manipulation by changing the individual pixels (bitmaps). The most powerful feature of this language is the ability to manipulate pixels of an image to perform tasks such as sharpening or blurring through matrix convolutions. Advanced techniques such as seam carving can also be applied to images using this language.

2 Lexical Conventions

There are six kinds of tokens: identifiers, keywords, constants, expression operators, and other separators.

2.1 Comments

The character sequence `:)` introduces a single line comment. Multi-line comments will be a sequence of single line comments using `:)`.

2.2 Identifiers

An identifier is a sequence of letters and digits that labels variables, functions, and classes; the first character must be alphabetic. The underscore _ and dash `-` are accepted in an identifier. Upper and lower case letters are considered different.

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

`return`	`boolean`	`break`
`def`	`null`	`throw`
`main`	`if`	`true`
`null`	`else`	`false`
`int`	`for`	`List`
`float`	`while`	`Pixel`
`string`	`continue`	`Image`

2.4 Literals

Literals are notations for constant values of some built-in types.

2.4.1 Integer literals

An integer literal is a sequence of digits, e.g `2`, `42`, `108`, or `-14`.

2.4.2 Floating point literals

A floating literal consists of an integer part, a required decimal point, and a fraction part. There can be no integer part or no fraction part, but at least one is required. The integer and fraction parts both consist of a sequence of digits, e.g. `1.0`, `.1`, `42.`, `1.`, or `-3.5`. A float literal must have a decimal point in order to be counted as a float; a literal without a decimal point like 1 or 42 will be interpreted as an integer.

2.4.3 Boolean literals

The boolean literal `true` is stored as a byte of value 1. The boolean literal `false` is stored as a byte of value of 0.

2.4.4 Character literals

A character literal is a single character surrounded by single quotes ' ' and stored as a 1-byte ASCII value.

2.4.5 String literals

A string is a sequence of characters surrounded by double quotes " ". In a string, the character " must be preceded by a backslash \; in addition, certain non-graphic characters, and \ itself, may be escaped according to the following table:

BS	\b
NL	\n
CR	\r
HT	\t
ddd	*ddd*
\	\\

2.4.6 Null literal

`null` is a special type with no name. You may not declare or cast a variable to `null` because it has no name. `null` may be ignored or considered as a special keyword. `null` may be compared to any type, and will only return `true` if compared to itself using `==`.

2.5 Delimiters

2.5.1 Parentheses

Parentheses ( ) are used in function declaration and calling, and in expressions to modify operator precedence. Conditionals and loops also require parentheses.

2.5.2 Curly braces

Curly braces { } are used to denote the start and end of a block of code; they are required after function declarations and at the beginning and end of each new block of code. They are also used when initializing an array.

2.5.3 Semicolon

A semicolon ; is required to terminate a statement. Any expression that is terminated with a semicolon will be executed on runtime. If this expression has no "side effect" such as assignment or function calling, then the compiler will throw an error. For loops will require semicolons in its syntax.

2.5.4 Commas

Commas , are used to separate expressions in function parameters, elements in list initialization, and multiple integer declarations either with the number of elements on the left equal to the number of elements on the right, or with only one element on the right e.g. a, b, c, = 1, 2, 3 OR a, b, c, = 5.

2.5.5 Periods

A period . is used to access a data member of a non-`List` data structure (e.g. `Pixel` or `Image`).

2.5.6 Whitespace

Whitespace is ignored in compilation and statements will be terminated only on semicolons, and not whitespace, tabs, or newlines. Each token in an expression is separated by these whitespace values.

3 Syntax and Semantics

3.1 Type specifiers

Types specifiers in declarations define the type of a variable or function declaration. To declare a variable, you must specify the type of the variable before its name, and to declare a function, you must specify its return type before its header. Examples of this can be seen below.

```
int x = 3;
def int myFunction(){}
```

3.2 Basic types

3.2.1 Integers

pixelman supports integers through the `int` type. An example of declaring and initializing an `int` variable:

```
int i = 42;
```

3.2.2 Floating points

pixelman supports single-precision, 32-bit floating point numbers through the `float` type. It is possible to assign integer values to a `float` type. An example of declaring and initializing a `float` variable through a `float` literal and an `int` literal:

```
float f = 42.42;
float f2 = -3;
```

3.2.3 Booleans

pixelman supports boolean values through the `boolean` type, and the `int` type. When evaluating integer values other than `true` and `false`, 0 evaluates to `false`, and any nonzero value evaluates to `true`. Trying to evaluate `null` as a `boolean` value results in an error. If a boolean evaluates to null, the compiler will throw an error. An example of declaring and initializing a `boolean`:

```
boolean isColor = true;
boolean isNonzero = 1;
```

3.2.4 Characters

pixelman supports chars which will be single characters surrounded by single quotes as seen in the examples below. Chars will be ASCII and have integer values ranging from 0-127.

```
char myLetter = 'a';
char myOtherLetter = 'b';
```

3.2.5 Strings

pixelman supports strings through the `string` type, which are stored as a `List` of `char` values.

3.2.6 Void

The `void` return type is only available to be used in functions that will not return any values.

```
def void functionName() {}
```

3.3 Lists

In pixelman, a `List` is an indexed container used to store multiple objects of the same type, with a static size and mutable elements. `List` elements are indexed beginning at position zero. To access the length of a `list`, perform the operation

```
lst.length; :) this will return an int value of the length of list lst.
```

3.3.1 Declaring lists

You declare a `List` by specifying the data type for its elements, the number of elements it can store, and its name. The number of elements must be a positive `int` value. For example, to instantiate a list of integers of length 3:

```
int[3] rgb;
```

This `List` will initialize with every element defaulting to 0.

3.3.2 Initializing lists

You can initialize the elements in a `List` when you declare it by enumerating the initializing values, separated by commas, in a set of curly braces. Here is an example of creating a `List` of the `int` values [255, 0, 0]:

```
int[3] rgb = {255, 0, 0};
```

When a `List` is initialized this way, either all elements or one element must be specified. If all elements are specified, then each element in the list will be assigned to its corresponding value. If only one element is specified, then every element in the list will be assigned to that value.

3.3.3 Accessing list elements

List elements will be accessed by writing the name of the list followed immediately by an open bracket, the number of the element, and a close bracket as seen in the example below.

```
int[3] rgb = {255, 0, 100};
int x = rgb[1]; :)x now has the value 0.
```

3.3.4 Multidimensional lists

You can make a multidimensional `List`, or a "list of lists", by adding an extra set of square brackets and list lengths for every additional dimension you wish your `List` to have.

```
int[3][3] mat = { {42, 0, 0}, {0, 42, 0}, {0, 0, 42}};
```

Multidimensional array elements are accessed by specifying the desired index of both dimensions.

```
mat[1][1] = 0;
```

3.4 Pixels and images

A pixel is a simple data type with five attributes:

- r for red value (RGB) (ranging from 0 to 255).
- g for green value (RGB) (ranging from 0 to 255).
- b for blue value (RGB) (ranging from 0 to 255).
- x representing the pixel's x, y coordinates.
- y representing the pixel's x, y coordinates.

A pixel may be initialized like this:

```
Pixel p = (r, g, b, x, y);
```

where r, g, b are the RGB values, and x, y are the pixel coordinates.

The syntax for accessing a pixel's properties:

```
p[0]; :) or p.r returns r value
p[1]; :) or p.g returns g value
p[2]; :) or p.b returns b value
p.x; :) returns x value
p.y; :) returns y value
```

An image is a 2-dimensional array of pixels that contain the image's height, width, and list of pixels. The user either loads an image from the file path or creates an empty image. To declare and initialize an image:

```
Image im = load("file_path");
Image im = new Image(h, w);
```

where $file\_path$ is the file path of the image, and h, w is the height and width. The latter will be an empty, white canvas of the designated height and width.

You may access the height and width:

```
im.height; :) the height, an integer
im.width; :) the width, an integer
```

3.5 Type conversions

pixelman will convert all operands in a mathematical equation to the largest operand implicitly. This means that any arithmetic operation with a float and an integer will return a float. The user may explicitly typecast on integers and floats, but may not explicitly convert lists, pixels, strings, characters or images.

To explicitly cast a type:

```
(<type>) variable;
```

3.6 Operators

3.6.1 Arithmetic

Addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`) work like regular arithmetic for types `int` and `float`. If applying any of these arithmetic operators on both an `int` and a `float`, the return type will be a `float`.

The double division sign (`//`) will divide and floor both arguments into integers.

The modulus operator (`%`) returns the remainder after dividing two arguments. The two arguments must be integers.

Some operations also work on 1 and 2 dimensional lists; their behavior will be described in section 3.7.

Addition, subtraction, multiplication, and division are illegal on string, boolean, and image types.

Addition and subtraction are illegal on pixels, but multiplication and division may be implemented. Multiplication and division will be scalar only, and changes the RGB values' brightness.

3.6.2 Comparison

The operator == is used to compare value of two operands of the same type. == is supported for int, float, boolean, string, and List. It does not allow for comparison between an int value and a float value. The operators >, <, >=, and <= are used to compare int values and float values, and also cannot compare values between the two.

```
bool a = 42 == 42; :) evaluates to true
bool b = 42 == 41; :) evaluates to false
bool c = 42 == 42.0; :) compiler will throw error

bool d = 42 < 43; :) evaluates to true
bool e = 42 > 43; :) evaluates to false
bool f = 42 <= 42; :) evaluates to true
bool g = 42 >= 42; :) evaluates to true
```

3.6.3 Boolean

The boolean operators !, &&, and || are supported for all types of operands except null.

```
bool a = ! true; :) evaluates to false
bool b = true && true; :) evaluates to true
bool c = true || false; :) evaluates to true
```

3.6.4 Bitwise

The operators << and >> can be used to bit shift ints left and right respectively.

```
int a = 4;
int b =  a << 2; :)evaluates to 16
int c = a >> 1 :)evaluates to 2
```

The operators &, |, and ^ can be used to represent the bitwise operations of and, or, and xor respectively.

```
int a = 4;
int b =  5;
int c = a & b :)evaluates to 4
int d = a | b :)evaluates to 5
int e = a ^ b :)evaluates to 1
```

3.6.5 Operator precedence

Statements with multiple operators in pixelman will follow the order of operations. This means that statements in parenthesis will be evaluated first. Multiplication and division will have the next highest precedence. Finally, addition and subtraction will be evaluated last. As an example, the statement below will evaluate to 61.

```
(4 + 5 * 5) + 8 * 4  :) evaluates to 61
```

3.7 Matrix/vector operations

The operations +, -, and * are supported for 1-dimension and 2-dimension lists of type int or float.

In order to perform these operations on lists, their dimensions must work for matrix or vector multiplication. The following cases for multiplying lists A and B are accepted:

- If A and B are 1-dimensional lists of length n,
  - A + B will return an n-length List of the sum of each element in its corresponding index.
  - A - B will return an n-length List of the result of A[i] - B[i] in index i.

- `A * B` will return a scalar (`int` or `float`) inner dot product of the two lists.

- If `A` is a `List` of dimension `a` by `b` and `B` is a `List` of dimension `c` by `d`,
  - `A + B` will return an `a` by `b` `List` of the sum of each element in its corresponding index if and only if both lists have the exact same dimension. If they have different dimensions, a runtime exception will be thrown.
  - `A - B` will return an `a` by `b` `List` of the result of `A[i][j] - B[i][j]` in index `i, j` if and only if both lists have the exact same dimension. If they have different dimensions, a runtime exception will be thrown.
  - `A * B` will return an `a` by `d` `List` of the matrix product of `A` and `B`, if and only if `b` = `c`. If `b` and `c` have different values, a runtime exception will be thrown.

  The type of the output is defined as such:

- If at least one operand is of type `float` and one is of type `float` or type `int`, the return type will be of type `float`.

- If both operands are of type `int`, then the return type will be of type `int`.

3.8 Assignment

There are 12 assignment operators; all are syntactically right-associative (they group right-to-left). Thus, a=b=c means a=(b=c), which assigns the value of c to b and then assigns the value of b to a.

Operators:

```
= *= /= %= += -= <<= >>= &= ^= |=
```

The result of the first operand of an assignment operator must be a variable, or a compile-time error occurs (cannot do something like 3=6). This operand may be a named variable, such as a local variable or a field of the current object, or it may be a computed variable, as can result from a field access or an array access.
The type of the assignment expression is the type of the variable.

3.9 Functions

3.9.1 Declaration

Functions are syntactically defined as:

```
def <type> <functionName>(arg1...argn) {
    <statement1>; <statement2>; ...; <statementn>;
    [return-statement;]
}
```

where `<type>` is the return type of the function, `functionName` is the unique label for a function, and `arg1...argn` are the optional arguments provided for the function.

3.9.2 Function calls

To execute a function, it must be called correctly with its unique name and the required parameters.
Function calls are syntactically defined as:

```
<functionName>(arg1...argn);
```

If a function has the incorrect number of arguments or an unrecognizable format, the line will return a compiler error.

3.10 Names and scope

Variables defined in a block of code will only be accessible in that block of code unless the declaration is preceded with the `global` keyword. If a variable is created with a name that already exists in that block of code, the compiler will throw an error.

3.10.1 Global variables

If a variable declaration is preceded with the `global` keyword, that variable will be accessible anywhere in that file. If there is another variable with the same name anywhere in the file, the compiler will throw an error.

3.11 Built-in Library

pixelman supports various functions through its standard library:

`int print(string format)` will print to standard out the designated string. On success it will return the number of characters written; on failure it will return -1.

`void perror(string format)` will print to standard error the user-defined string.

`int scan(string buffer)` will scan from standard a string. On success, it will return the number of characters scanned; on failure, it will return -1.

`int size(List arr)` will get the length of the list. On success, it will return the int value of the list size; on failure, it will return `-1`.

`Image load(string file_path)` will load an image from a file path. On success, it will return an image; on failure, it will return `null`.

`int write(string file_path, Image im)` will write an image to a file path. On success, it will return 0; on failure, it will return `-1`. If the file path already has something there, it will create a copy and create a new file automatically.

`int display(Image im)` will display the image without saving to disk. On success, it will return 0; on failure, it will return `-1`.

`image resize(Image im, int height, int width)` will resize an image to the provided height and width. On success, it will return image; on failure, it will return `null`.

`image transform(Image im, processingFunction())` will perform an operation defined by the user in `processingFunction()` on each pixel in image. It will create a new image. On success, it will return image; on failure, it will return `null`.

4 Statements

4.1 If.. else

pixelman supports `if/else` statements that allow conditional boolean statements to control the execution flow of the code. Conditional `if` statements can be nested multiple times. An `if` statement may be followed by an `else` that will execute if the condition in the if statement evaluates to false.

```
def <type> <functionName>(arg1...argn){
    if(<booleanExpression) {
        :) code
    } else {
        :) code
    }
}
```

4.2 Loops

4.2.1 Breaks

The `break` statement will terminate the execution of the nearest loop, and will resume to the next statement following the loop. It will essentially "jump out" of the loop.

```
for(<start>; <booleanExpression>; <newAssignment>) {
    if(<booleanExpression2>) {
```

```
            break;
            :) will terminate the for-loop
        }
    }
```

4.2.2 Continue

The `continue` statement will break one of the iterations of the loop, and will continue to the next iteration of that loop. It will essentially "jump over" one iteration.

```
for(<start>; <booleanExpression>; <newAssignment>) {
    if(<booleanExpression2>) {
        continue;
        :) will terminate this iteration without doing anything
        :) and go on to the next iteration
    }
}
```

4.2.3 For loops

pixelman supports `for` loops that will run a block of code for as long as the conditional statement holds. The `for` loop will have a starting point `<start>`, typically an assignment to a variable, and that starting point will change according to the `<newAssignment>`. Once changed to `<newAssignment>`, the for loop will evaluate the `<conditionalStatement>` once again, and execute the function if true. Otherwise, it will exist the `for` loop. `for` loops may be nested multiple times.

```
def <type> <functionName>(arg1...argn){
    for(<start>; <booleanExpression>; <newAssignment>) {
        :) code
    }
}
```

4.2.4 Enhanced For Loops

Enhanced For Loops iterate through a list containing objects of a specific type.

```
            for(<item> in <List>){
            ...
            ..
            .
            }
```

Type of <item> is of same type contained in <List>.

4.2.5 While loop

pixelman supports `while` loops that will run a block of code as long as the condition in the `while` evaluates to true.

```
def <type> <functionName>(arg1...argn){
    while(<booleanExpression>) {
        :) code
    }
}
```

4.3 Blocks

Blocks in pixelman will begin with a "{" and end with a "}." Blocks can be nested inside of each other. A "}" will mark the end of the block that began with the most recent "{" that has not yet been closed. Blocks can only be used following function declarations, `if` statements, `else` statements, `for` loop declarations, and `while` loop declarations.

4.4 Return

Return statements are defined as:

```
return <something>;
```

The return type must match the type explicitly stated in the function declaration. If it does not match the type, it will throw an error.

5 Formal Grammar

1. Expressions

expression:
primary
`!` *expression*
expression binop expression
expression `,` *expression*

primary:
identifier
constant
string
`(` *expression* `)`
primary `[` *expression* `]`
lvalue `.` identifier

lvalue
identifier
primary `[` *expression* `]`
lvalue `(` identifier
`(` *lvalue* `)`

The primary-expression operators
`()` `[]` `.`
have highest priority and group left-to-right. The unary operator ! has priority below the primary operators but higher than any binary operator, and group right-to-left. Binary operators all group left-to-right, and have priority decreasing as indicated:

binop:
`*` `/` `%` `//`
`+` `-`
`<<` `>>`
`<` `>` `<=` `>=`
`==` `!=`
`&`
`^`
`|`
`&&`
`||`

Assignment operators all have the same priority, and all group right-to-left.

asgnop:
`=` `=`

The comma operator has the lowest priority, and groups left-to-right.

2. Declarations

declaration:
type-specifier declarator-list$_{opt}$;

type-specifier:
`int`
`float`
`char`
`bool`
`string`

declarator-list:
declarator
declarator, declarator-list

declarator:
identifier
declarator ( )
declarator [*constant-expression*$_{opt}$]
(*declarator*)

type-decl-list:
type-declaration
type-declaration type-decl-list

type-declaration:
type-specifier declarator-list ;

3. Statements
statement:
expression ;
{*statement-list*}
`if`*(expression) statement*
`if` (*expression*) *statement* `else` *statement*
`while`*(expression) statement*
`for`*(expression1 ; expression2; expression3) statement*
`for`*(expression1 : dataType) statement*
`continue;`
`break;`
`return;`
`return(`*expression*`)`
`;`

statement-list:
statement
statement statement-list

6 Sample Program

Included is a program to read in an image from a filepath, transform it into a greyscale version, and display it

```
def Image greyscale(String filePath) {
    if(Image im = load(filePath) == null) {
        perror("File path is not found.");
    }

    print("Original image:");
    display(im);
```

```
    for(int i = 0; i < im.height; i++) {
        for(int j = 0; j < im.width; j++) {
            :) rough formula for greyscale = average of RGB values
            int greyscale = (im[i][j][0] + im[i][j][1] + im[i][j][2]) / 3;
            for(int k = 0; k < 2; k++) {
                im[i][j][k] = greyscale;
            }
        }
    }

    printf("Greyscale image:");
    display(im);
    return(im);

}


def int main(String[] args) {
    String filePath = "filePath/pixelman/example.bmp"
    Image newImage = greyscale(filePath);

    :) change to square
    if(newImage.height != newImage.width) {
        resize(newImage, height, height);
    }

    write(filePath, newImage);

    return 0;
}
```