Pixel

Language Reference Manual

Alex Anthony Cortes-Ose (ac4441) - Language Guru
Dillon Davis (dhd2121) - Manager
Jessica Kim (sk4711) - Tester
Jessica Peng (jp3864) - System Architect

Table of Contents

1: Introduction

Originally inspired by the concept of applying filters to images in social media platforms like Snapchat and Instagram, Pixel is a language designed to process and manipulate images. With Pixel, it is easy to write algorithms that perform enhancements, transformations, edge detection, and basic analysis on images. An *image* is its own primitive type, containing a *pixel* matrix; *pixel* is another primitive type. These types enable extended functionality and increased efficiency in many image processing problems and applications.

The syntax for Pixel draws from Python and JavaScript with their effectiveness for array and matrix manipulation.

2: Data Types

2.1 Primitives

Context-free grammar:
type-specifier-decl

Data Type	Description
`int`	32-byte signed integer type
`float`	64-byte floating point number
`str`	Array of ASCII characters

2.2 Structures

Structured data types are dynamically sized and mutable.
Context-free grammar:
struct::type-specifier-decl

<table>
<tr><th>Structured Data Type</th><th colspan="2">Description</th></tr>
<tr><td><code>dict::key_type, value_type</code></td><td colspan="2">Hash table storing <code>int</code>, <code>float</code>, or <code>str</code>. Follows syntax:
<code>{ “a”: 1, “b”: 2, “c”: 3, “d”: 4 };</code></td></tr>
<tr><td><code>list::type</code></td><td colspan="2">A standard array consisting of elements of the same type. Follows syntax: <code>[“a”, “b”, “c”];</code></td></tr>
<tr><td rowspan="3"><code>matrix::type</code></td><td colspan="2">Mutable matrix data structure storing 2 or 3 dimensions of <code>int</code>s or <code>float</code>s. Follows syntax:
<code>[[3 0 0]
[10 1 -3]
[-2 9 0]];</code></td></tr>
<tr><td>Function</td><td>Description</td></tr>
<tr><td><code>transpose()</code></td><td>Transpose a given <code>matrix</code> flipping</td></tr>
</table>

		rows and cols dimensions Returns a `matrix`
	`rows(), cols()`	Return the number of rows and columns in a given `matrix` Returns an `int`

2.3 Image and Pixel

The image and pixel are specialized data types which expose useful functionality for performing various operations relevant to the field of image processing.

Context Free Grammar:

type direct-declarator (parameter-type-list)

Data Type	Description	
`image`	A `matrix` wrapper with built-in functions. Represents an image as a 3-D matrix of pixels, integer brightness values, or floating point color or brightness values	
	Function	Description
	`convert("grayscale" || "color" || "double")`	Convert a given image to either a different color space or the current colorspace representing color values as double precision floating point numbers Returns an `image`
	`red(), green(), blue()`	Extract the corresponding red, green, or blue channel from a given image Returns an `image`
	… matrix `rows()` *and* `cols()` *can be applied to an image*	
`pixel`	A `list` wrapper on top of a 3 element long list of either `int`s or `float`s with built-in functions. Represents an image pixel	
	Function	Description
	`red(), green(), blue()`	Extract corresponding color value (the underlying list can also be indexed) Returns an `int` or `float`
	`brightness()`	Extract corresponding pixel brightness / grayscale value Returns an `int` or `float`

3: Lexical Conventions

3.1 Keywords

The following are reserved keywords for the Pixel language – all data types are considered as reserved keywords as well.

Context Free Grammar:

selection-statement:

if *(expression) statement*

if *(expression) statement* else *statement*

while *(expression) statement*

return *expression (IF NOT VOID) **if not executed, implicit return with no expression***

Keyword	Description
`if` `else`	Standard if, else clause. Follows syntax: `if (condition) {statements} else {statements}`
`while`	Standard while loop that executes statements while a condition is true. Follows syntax: `while (condition) {statements}`
`fun`	Function declaration. Follows syntax: `fun::type function_name (type var_name, … ) { … }`
`return`	Halts the current function execution and returns a value. Follows syntax: `return var_name or expression;`
`void`	Indicates that a function has no return value
`print`	Function that prints any data type to standard output

3.2 Operators

Pixel enables many different ways to utilize basic arithmetic and matrix operators with support for different operand types. Below is a detailed table of supported operand types and a description of the expected computation. Here, `scalar` refers to either `int` or `float`, and `matimg` refers to either `matrix` or `image`.

Operator and Operands		Description
`+`	`scalar + scalar -> scalar`	Standard scalar addition, returns sum
	`matimg + scalar -> matimg`	Increases each `matrix` element by a `scalar`, returns a scaled `matrix`
	`matimg + matimg -> matimg`	Element-wise `matrix` addition, returns the `matrix` of summed elements

-	`scalar - scalar -> scalar`	Standard `scalar` subtraction, returns difference
	`matimg - scalar -> matimg`	Decreases each `matrix` element by a `scalar`, returns a scaled `matrix`
	`matimg - matimg -> matimg`	Element-wise `matrix` subtraction, returns the `matrix` of subtracted elements
* and .*	`scalar * scalar -> scalar`	Standard `scalar` multiplication, returns product
	`matimg * scalar -> matimg`	Multiplies each `matrix` element by a `scalar`, returns a scaled `matrix`
	`matimg * matimg -> matimg`	Returns the dot product of two `matrices`
	`matimg .* matimg -> matimg`	Element-wise `matrix` multiplication, returns the `matrix` of products
/ and ./	`scalar / scalar -> scalar`	Standard `scalar` division, returns the quotient
	`matimg / scalar -> matimg`	Divides each `matrix` element by a `scalar`, returns a scaled `matrix`
	`matimg ./ matimg -> matimg`	Element-wise `matrix` division, returns the `matrix` of quotients
%	`scalar % scalar -> scalar`	Returns the remainder of standard `scalar` division
>	`scalar > scalar -> 1 or 0`	Returns 1 if a `scalar` is greater than another `scalar`, 0 otherwise
>=	`scalar >= scalar -> 1 or 0`	Returns 1 if a `scalar` is greater than or equal to another `scalar`, 0 otherwise
<	`scalar < scalar -> 1 or 0`	Returns 1 if a `scalar` is less than another `scalar`, 0 otherwise
<=	`scalar <= scalar -> 1 or 0`	Returns 1 if a `scalar` is less than or equal to another `scalar`, 0 otherwise
==	`scalar == scalar -> 1 or 0`	Returns 1 if a `scalar` is equal to another `scalar`, 0 otherwise
!=	`scalar != scalar -> 1 or 0`	Returns 1 if a `scalar` is unequal to another `scalar`, 0 otherwise

An `image` contains three color channels which each can be accessed and then manipulated via `matrix` operators. Each of the above `matrix`-by-`matrix` and `matrix`-by-`scalar` operations also applies to `image`s returning either an `image` or a `scalar`, respectively.

```
image base = image_in("../imgs/base_img0.png");
image boosted_red = base.red() + 100;
image_out("boosted_red", join(boosted_red, base.green(), base.blue()));
```

3.3 Literals

Literal Type	Description	Example
Integer Literals	A series of digits that represent a number	`0, 1, 2, …, 9`
Float Literals	A series of zero or multiple digits accompanied by a dot character after the first sequence, followed by 1 or more digits	`12.34`
String Literals	A series of character primitives encompassed by quotation marks that represents a string that is unnamed	`"Hello World"`

3.4 Identifiers

Identifiers follow the same "lowercase letters separated by underscores" convention as is used in the Python language. They must begin with a lowercase letter, and can be followed by any combination of lowercase and uppercase letters, numbers, or underscores.

```
int my_varNAME = 0;
fun::image threshold_1(image default) { /* code here */ }
str new_image_name = "basic_blur.png";
```

3.5 Comments

Single line comment: `//`
Multi line (nestable) comment: `/* */`

3.6 Variables

Variables that are uninitialized outside the function must be declared at the start of the program, they can be assigned values inside function bodies. Any variable declarations inside functions must occur at the start of the function scope.

4: Syntax

4.1 Variables

Variables are defined by first specifying a type and then a valid identifier. They are assigned with the identifier followed by the assignment (=) character within a function body. Structured types including `matrix`, `list`, and `dict` must specify stored types after their double colon characters.

```
str new_image_name;
int example;

fun::void main() {
  dict::str,str props;
  matrix::int identity_kernel;

  identity_kernel = [
  [0 0 0]
  [0 1 0]
  [0 0 0]
];
  props = { “name”: “John”, “date”: “02-23-21” };
  new_image_name = “basic_blur.png”;
  example = 0;
}
```

4.2 Functions

Functions are defined with the `fun` keyword and the return type must be specified after double colon characters. Arguments must also be typed, and statements for the function body are written within a pair of curly braces.
Context Free Grammar:
key::type-decl
key::type-decl-list

```
fun::int add_two(int a, int b) {
  return a + b;
};
```

4.3 Expressions

When an expression contains multiple operators, these operators are executed based on rules of precedence. The following lists the types of expressions in order of highest to lowest precedence. For operators of equal precedence, they are executed from left to right.

1. Function calls, array subscripting, and membership access-operator expressions
2. Unary operators: logical negation
3. Multiplication, division, and modular division expressions
4. Addition and subtraction (scalar and matrix) expressions
5. Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to (scalar and matrix) expressions
6. Equal-to and not-equal-to expressions
7. Logical AND expressions
8. Logical OR expressions
9. All assignment expressions
10. Comma operator expressions

4.4 Array and Matrix Access

The square brackets allow the user to access a specific element within an array with an integer index, and within a `matrix` with comma-separated integers wrapped by square brackets.

```
/* Accessing an element within an array */
list::int array = [1,2,3];
int first_element = array[0];

/* Accessing an element within a matrix */
matrix::int m = [
  [1 0 3]
  [2 4 9]
  [0 0 0]
];
int matrix_element = m[1, 2]; /* This will be the value 9 */
```

5: Standard Library

Context Free Grammar:

Type conversion explicitly by cast: *type-name specifier-qualifier-list abstract-declarator*

Function	Description
`len(list l)`	Takes one argument of type `list` and returns the length
`range(int start, int end)`	Takes two arguments of type `int` and returns the range from start inclusive to end exclusive

`sum(list l)`
`or`
`sum(matrix m)`
	Returns the sum of all elements in a `list` or the element-wise sum of a `matrix`

`image_in(str file_path)`	Takes one argument of type `str` which specifies the file

	path of a png image and returns an `image`

`image_out(str file_name, image i)`
or
`image_out(str file_name, matrix m)`
	Takes two arguments of type `str` and either `image` or `matrix` and saves the second argument as a png image to the user's hard disk

`join(image r, image g, image b)`
or
`join(matrix r, matrix g, matrix b)`
	Takes in three arguments of type `image` or `matrix`. Combines each argument as respective red, blue, and green channels and returns a color `image`

`zeros(int m, int n)`	Takes in two arguments of type `int`, called m and n, and returns an mxn `matrix` filled with zeros

`int(float f)`
or
`int(str s)`
	Casting a `float` or `str` into an `int`

`float(int i)`
or
`float(str s)`
	Casting an `int` or `str` into a `float`

`str(int i)`
or
`str(float f)`
	Casting an `int` or `float` into a `str`

`matrix(list l)`	Casting a `list` into a `matrix`
`image(matrix m)`	Casting a `matrix` into an `image`

6: Code Samples

6.1 Thresholding

```
fun::image binary_brightness_threshold(image in) {
  matrix::int m = zeros(in.rows(), in.cols());
  for (int i = 0; i < in.rows(); i++) {
    for (int j = 0; j < in.cols(); j++) {
      m[i, j] = in[i, j] >= 178;
    }
```

```
  }
  return image(m);
}
```