PixMix

Language Reference Manual

Table of Contents

Lexical Elements

Identifiers

An identifier is a case-sensitive sequence of alphanumeric characters. The first character must be alphabetic or an underscore.

Keywords

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

```
String
Image
Pixel
Color
Console
for
while
break
continue
char
int
float
return
array
Object
bool
true
false
if
elif
else
switch
case
and
or
not
fun
```

Constants

Constants are global variable declarations that may not be assigned new values.

Integer Constants

An integer constant is a sequence of digits that is globally available within a program.

```
INT_MAX
FLOAT_MAX
```

Characters and Character Arrays

PixMix uses the ASCII character set for its implementation of characters.

A literal of type character consists of a single character or escape sequence inside two single quotes :

```
‘c’ or ‘\n’
```

String Constants

In PixMix, Strings are a standard library that include many string manipulation functions.

Floating Point Constants

A floating point constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. 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.

Separators

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

```
( ) [ ] { } , . : ;
```

White Space

White space is the collective term used for several characters: the space character, the tab character, the newline character, and the carriage-return character. White space is ignored (outside of string and character constants), and is therefore optional, except when it is used to separate tokens.

Data Types

Each primitive type, integer, float, string, boolean, character, byte, is indicated by a name that PixMix uses for declarations.

`int` A 32-bit, signed, 2's complement series of digits with a maximum range of 2147483647.

`float` Follows IEEE 754-2008's definition of a floating-point number. That is: 1 bit for the sign, 8 bits for the exponent, and 24 bits for the mantissa.

`bool` A 1 byte value that can store either 1 or 0, or true or false.

`char` A character is 8-bit (1 byte) data type, capable of holding one character in the local character set.

Primitive Data Types

Integer Types

`unsigned char`

`int`

Real Number Types

float

Objects

To define an object, use the `Object` keyword followed by the name of the object. An object may be initialized during instantiation by specifying a list of its member variables and functions within curly brackets.

```
Object rectangle = {
    int width, height
    fun area() {
            return width * height
    }
}
```

Arrays

Array elements are indexed beginning at position zero.

Declaring Arrays

You declare an array by specifying the data type as Array, followed by its name. The array can store multiple data types. An example declaration:

```
Array myArray;
```

Initializing Arrays

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

```
Array myArray = [0, “one”, 2, “three”]
```

Accessing Array Elements

You can access an element in an array by specifying the array name followed by the element index, enclosed in brackets.

```
myArray[1] = 1
```

This will assign the value 1 to the second element in the array, at position one.

Multidimensional Arrays

Arrays can contain another array as an element, creating a multidimensional array.

```
Array a = [[1,2,3], [2,3,4]]
```

Elements will then be accessed by specifying first the index of the nested array, and second the index of the element within the nested array.

```
a[1][2]
```

Would return 4.

Expressions and Operators

Expressions

Assignment Operators

- +=
    - Adds the two operands together, and then assign the result of the addition to the left operand.
- -=
    - Subtract the right operand from the left operand, and then assign the result of the subtraction to the left operand.
- *=
    - Multiply the two operands together, and then assign the result of the multiplication to the left operand.
- /=
    - Divide the left operand by the right operand, and assign the result of the division to the left operand.
- %=
    - Perform modular division on the two operands, and assign the result of the division to the left operand.
- <<=
    - Perform a left shift operation on the left operand, shifting by the number of bits specified by the right operand, and assign the result of the shift to the left operand.
- >>=
    - Perform a right shift operation on the left operand, shifting by the number of bits specified by the right operand, and assign the result of the shift to the left operand.
- &=
    - Perform a bitwise conjunction operation on the two operands, and assign the result of the operation to the left operand.
- ^=
    - Performs a bitwise exclusive disjunction operation on the two operands, and assign the result of the operation to the left operand.
- |=
    - Performs a bitwise inclusive disjunction operation on the two operands, and assign the result of the operation to the left operand.

Incrementing and Decrementing

The increment operator ++ adds 1 to its operand. The operand must be either an int, float, Image or Pixel. If the ++ is before the operand as in `++i * 5`, `++i` is incremented before the other operation. If the increment operator is after such as in `i++ * 5`, then the increment happens after.

Arithmetic Operators

The following arithmetic operators are utilized in PixMix

`+`	`addition`
`-`	`subtraction`
`*`	`multiplication`
`/`	`division`
`%`	`modulo`
`++`	`prefix and postfix incrementation`
`--`	`prefix and postfix decrementation`
`()`	`control arithmetic preference`

Comparison Operators

For comparing equivalent values, the keyword **`is`** is used.
Conversely, to determine if two values are not equal, the keyword **`not`** is used.
Greater than, less than, and their variants use standard symbols (<, >, <=, >=).

```
int x = 1;
int y = 3;

if x is y
   Console.log(“X is the same as Y”);
if x not y
   Console.log(“X and Y are not the same”);
if x <= y
   Console.log(“X is less than Y”);
```

Logical Operators

The conjunction operator in PixMix is the keyword **and** while the disjunction operator is the keyword **or**. To negate a logical expression, use the keyword **not**. Logic can be nested using parentheses.

```
if x is 5 and y > 3
   Console.log(“X is 5 and Y is greater than 3”);
if x is 5 or y is 3
   Console.log(“X is 5 or Y is 3”);
if x is 5 and (y is 3 or z is 5)
   Console.log(“X is 5 and either Y is 3 or Z is 5”);
```

Bit Shifting

The left-shift operator << is used to shift its operand’s bits to the left, while the right-shift operator >> shifts to the right. The second operand denotes the number of bit places to shift by. Bit shifted off the left or right sides are discarded.

```
int x = 47;    // x is 00101111 in binary
x << 1;        // x is 01011110 in binary, or 94 in decimal
x >> 1;        // x is 00101111, or 47, again
```

Bitwise Logical Operators

& Conjunction -- When both bits are 1, the result is 1, otherwise it’s 0.

```
10110 & 10101 = 10100
```

| Inclusive Disjunction -- When both bits are 0, the result is 0, otherwise it’s 1.

```
10110 | 10101 = 10111
```

^ Exclusive Disjunction -- When bits are different, the result is 1, otherwise it’s 0.

```
10110 & 10101 = 00011
```

~ Negation -- Reverses each bit, 1s become 0s, 0s become 1s.

```
~001011 = 110100
```

}

Type Casts

You can use a type cast to explicitly cause an expression to be of a specified data type. A type cast consists of a type specifier enclosed in parentheses, followed by an expression. To ensure proper casting, you should also enclose the expression that follows the type specifier in parentheses.

```
float x;
int y = 7;
int z = 3;
x = (float) (y / z);
```

Member Access Expressions

You can use the member access operator . to access the members of a structure or union variable. 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;
```

Conditional Expressions

Pixmix uses a Java-like ternary operator. The following code:

```
int x;


if(conditionIsTrue) {
   x += 5;
} else {
   x = 0;
}
```

can be written using the conditional operator as:

```
 int x = (conditionIsTrue) ? x + 5 : 0;
```

Operator Precedence

When an expression contains multiple operators. The operators are grouped based on rules of *precedence*. The order of precedence in PixMix follows the order of operations in C. Notably, PixMix evaluates from right to left when multiple assignment statements appear as subexpressions in a single larger expression.

The following is a list of types of expressions, presented in order of highest precedence first. Sometimes two or more operators have equal precedence; all those operators are applied from left to right unless stated otherwise.

1. Function calls, array subscripting, and membership access operator expressions.
2. Unary operators, including logical negation, bitwise complement, increment, decrement, unary positive, unary negative and type casting. When several unary operators are consecutive, the later ones are nested within the earlier ones: !-x means !(-x).
3. Multiplication, division, and modular division expressions.
4. Addition and subtraction expressions.
5. Bitwise shifting expressions.
6. Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to
7. expressions.
8. Equal-to and not-equal-to expressions.
9. Bitwise AND expressions.
10. Bitwise exclusive OR expressions.
11. Bitwise inclusive OR expressions.
12. Logical AND expressions.
13. Logical OR expressions.
14. Conditional expressions (using ?:). When used as subexpressions, these are evaluated right to left.
15. All assignment expressions, including compound assignment. When multiple assignment statements appear as subexpressions in a single larger expression, they are evaluated right to left.

Statements

Labels

Expression Statements

Most expression statements are variable assignment or functions assignments of the form:

```
expression;
```

The if Statement

You can use the `if` statement to conditionally execute part of your program in PixMix, based on the truth value of a given expression. Here is the generalized form of an `if` statement:

```
   if expression
     statement;
   else
     statement;
```

If test evaluates to true, then `then-statement` is executed and `else-statement` is not. If test evaluates to false, then `else-statement` is executed and `then-statement` is not.

The elif Statement

The elif statement works the same as “else if” in C. In the case where the user wants to specify a series of boolean checks and actions they can use elif as follows:

```
if expression

    statement;

elif expression

    statement;
else
    statement;
```

The switch Statement

You can use the switch statement in PixMix to compare one expression with others, and then execute a series of sub-statements based on the result of the comparisons.

The while Statement

Within PixMix, the `while` statement is a loop statement with an exit test at the beginning of the loop. If the test evaluates true then the statement is executed. The statement continues to be executed as long as the test evaluates to true.

The for Statement

PixMix supports `for` loop statement for repeated code execution as well. A for loop is used to iterate over the values of an `Array`, `object`, or `struct` that has iterable values. The loop will continue until every element in the given object has been used or the loop encounters a `break` or `continue`.

```
for value in object
```

Where `value` is a temporary variable that is assigned the first value that exists in `object`. When the loop repeats, `value` will hold the second value in `object`, and so on.

The break Statement

You can use the `break` statement to terminate a `while`, `for`, or `switch` statement. If you put a `break` statement inside of a loop or `switch` statement which itself is inside of a loop or `switch` statement, the `break` only terminates the innermost loop or `switch` statement.

The continue Statement

You can use the `continue` statement in loops to terminate an iteration of the loop and begin the next iteration. If you put a `continue` statement inside a loop which itself is inside a loop, then it affects only the innermost loop.

Comments

Single-line comments in PixMix start with `//` and end at the end of the line. Pixel also supports multi-line comments which start with `/*` and end with the following `*/`.

Functions

Function Definitions

A function is defined via the keyword **`fun`**. The optional parameters to be passed into the function are defined along with their types in parenthesis. A function's body must be enclosed with curly brackets, no matter its length.

```
fun myFunc(int a, int b) {
   // Function
   // Body
}
```

Values are returned using the `return` statement.

Calling Functions

A call to any function which returns a value is evaluated as an expression.

```
fun function(void) {
   return 3
}
int a = 10 + function() // a is 13
```

Function Parameters

Function parameters can be any expression—a literal value, a value stored in variable, a function call, or a more complex expression built by combining these.

Recursive Functions

Recursive functions are functions that call themselves. This example of calculating the factorial using a recursive call illustrates how recursion is implemented in PixMix:

```
int factorial (int x)
{
   if (x < 1)
         return 1
   else
```

```
        return (x * factorial (x - 1))
}
```

Standard Libraries

PixMix includes several libraries that include a variety of utility functions.

Image

The Image object holds an array of pixels. It supports the +, - and % operands.

Adding two images will result in an image which pixels are the averages of the pixels of both images. That is, each hex value in the RGB of the pixel will be added and divided by 2.

```
Image newImage = img1 + img2
```

Subtracting two images will result in the difference between each pixel in the two images.

Dividing two images with one another will split the two images in half vertically and create and image that contains the lefthand image on the left and the righthand image on the right.

```
Image smileySad = smiley % sad
```

Pixel

A Pixel stores an RGB value in hex. Pixels can be incremented per Red, Green, Blue value as follows.

```
Pixel p = [“E2”,”72”,”5B”]

int i = 0
while i < 0 {
    p.blue++
    i++
}
```

Color

The standard library contains a set of predefined colors which are just pixels with prestored RGB values.

The colors in the standard library are:

red
green
blue
yellow
brown
orange
white
black
gray
purple
maroon
terracotta
lime
indigo

A color can be assigned to a pixel:

```
Pixel p = lime
```

String

The string object uses a character array under the hood and allows the user to concatenate two strings by using the += operator.

```
String a = “ball”
+= “oon”
//results in “balloon”
```

The String object also supports the + operand.

Program Structure and Scope

Program Structure

An entire program written in PixMix may be within a single source file, or may be broken up into several other files and included when necessary.

Scope

In PixMix, declarations made at the top-level of a file (i.e., not within a function) are visible to the entire file, including from within functions, but are not visible outside of the file. Any variable delcared outside of brackets is available everywhere in that file. Declarations made within functions are visible only within those functions.A declaration is not visible to declarations that came before it.

Control Flow

Control is managed by indentation.

```
if expression:
    statement
```

Sample Programs

Hello World

```
// prints greeting according to time of day
String out = “Good”;

// Assuming a standard library Time exists
if Time.now < Time.get(1200)
   out += “ morning!”
elif Time.now < Time.get(“5 PM”)
   out += “afternoon!”
else
   out += “evening!”

Console.log(out) // Print the contents of “out” to stdout
```

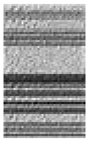
Blacken any pixels that are “too” red, blur the image, then save as a new file

```
// Load an image from disk
Image img = Image.load(“sample.bmp”)

// Loop over every Pixel in the Image
for p in img
   // If red channel is more than 100, remove red
   if p.red > 100
         p.red = 0

img.gaussianBlur(3); // Blur the image
img.saveAs(“sample-redMute-Blur.bmp”); // Save to a new file
```

Fibonacci

```
fun fib(int i) {
   if i <= 1
         return i
   return fib(i-1) + fib(i-2)
}

int i
Console.read(i)      // Prompt user for input
Console.log(fib(i))  // Print result to stdout
```

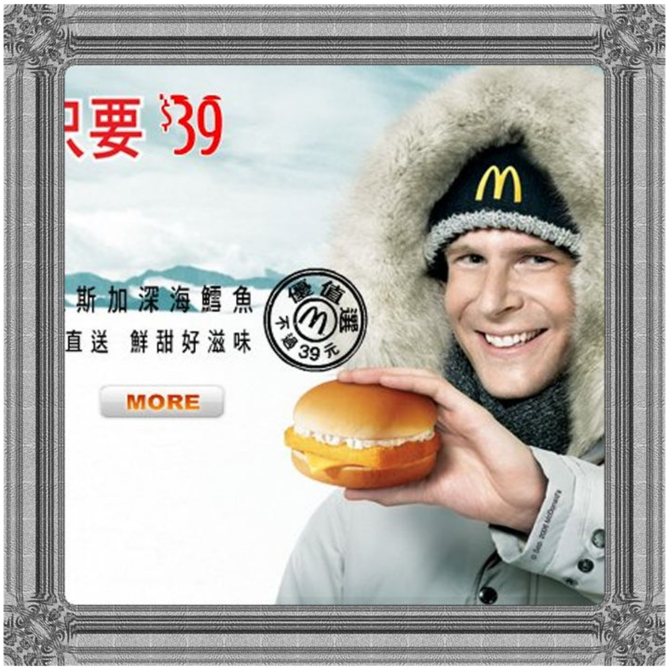
Compositing Images

Corner	Frame	Base
		只要 $39 阿拉斯加深海鱈魚 產地直送 鮮甜好滋味 優值選 不過39元 MORE 鱈魚製成，供應來源會依季節性而不同，但一樣保留整片鱈魚的鮮甜美味

```
    Image base = Image.load(“base.bmp”)
    Image corner = Image.load(“corner.bmp”)
    Image frame = Image.load(“frame.bmp”)

    int frameRepX = base.width / frame.width
    int frameRepY = base.height / frame.height

    for i in frameRepX {
        // Use the Image object’s place function to add the image called
“frame”
        // onto the image called “base”
        // destination.place(source, x, y)
        base.place(frame, i, 0)
        base.place(frame, i, base.height)
    }
```

```
// Rotate the image called “frame” by 90 degrees in preparation for
placing it along the sides of the image called “base”
frame.rotate(90);
for i in frameRepY {
   base.place(frame, i, 0)
   base.place(frame, i, base.width)
}



base.place(corner, 0, 0)
corner.rotate(90)
base.place(corner, base.width, 0)
corner.rotate(90)
base.place(corner, base.width, base.height)
corner.rotate(90)
base.place(corner, 0, base.height)

base.saveAs(“SAEFramed.bmp”)
```

Result, SAEFramed.bmp



Grammar

```
program:
    declaration_list

declaration_list:
    declaration_list declaration
    declaration

declaration:
    variable_declaration
    function_declaration

variable_declaration:
    type identifier = expression
    type identifier

function_declaration:
    fun identifier (parameter_list) { statement_list }

parameter_list:
    /* nothing */
    parameter_list type identifier

statement_list:
    /* nothing */
    statement_list statement

statement:
    assign_statement
    break_statement
    compound_statement
    expression_statement
    iteration_statement
    return_statement
    selection_statement
```

```
assign_statement:
    identifier = expression

break_statement:
    break

compound_statement:
    statement_list

expression_statement:
    expression

iteration_statement:
    while expression statement
    while (expression) statement
    for constraint_list statement
    for (constraint_list) statement

return_statement:
    return optional_expression

selection_statement:
    if expression optional_selection_statement_list statement
    if expression
    if expression optional_selection_statement_list else statement

optional_selection_statement_list:
    optional_selection_statement_list else if expression statement
    else if expression statement

optional_expression:
    /* nothing */
    expression
```

```
constraint_list:
    constraint
    constraint_list, constraint

constraint:
    type identifier in identifier

expression:
    mutable = expression
    mutable += expression
    mutable −= expression
    mutable ∗= expression
    mutable /= expression
    mutable ++
    mutable −−
    simpleExpression

mutable:
    identifier

simple_expression:
    logical_or_expression
    expression logical_or_expression

logical_or_expression:
    logical_and_expression
    logical_or_expression or logical_and_expression

logical_and_expression:
    equality_expression
    logical_and_expression and equality_expression

equality_expression:
    relational_expression
    equality_expression is relational_expression
    equality_expression is not relational_expression
```

```
relational_expression:
    additive_expression
    relational_expression < additive_expression
    relational_expression > additive_expression
    relational_expression <= additive_expression
    relational_expression <= additive_expression

additive_expressions:
    multiplicative_expression
    additive_expression + multiplicative_expression
    additive_expression - multiplicative_expression

multiplicative_expression:
    image_expression
    multiplicative_expression * image_expression
    multiplicative_expression / image_expression
    multiplicative_expression % image_expression
    multiplicative_expression & image_expression

image_expression:
    unary_expression

unary_expression:
    primary_list
    unary_operator primary_list

unary_operator:
    (type)
    &
    &=
    |
    |=
    ^
    ^=
    <<
    <<=
    >>
```

```
>>=

primary_list:
    atoms
    index
    call

index:
    identifier[expression]

call:
    identifier (actual_list)

actual_list:
    expression
    actual_list

atoms:
    literals
    identifier
    (expression)

literals:
    character_literal
    integer_literal
    float_literal
    true
    false
```

Appendix

Syntax Quick-Reference

General

No semicolons

No “main” function

Use curly brackets around multiline statements

No curly brackets around single line statements

EXCEPTION: Objects always use curly brackets

Functions

Use parentheses around parameters

Conditionals

No parentheses around the expression (unless needed for precedence)

Comments

C style: // single line   /* block */

Array

Array myArray = [1,2,3]

Data type, name, equals, square brackets around comma separated list

Object

Object myObject = { int x}

Data type, name, equals, curly brackets around statements.

PixMix Team

Name	**Uni**	**Role**
Alexandra Taylor	at3022	Tester
Christina Charles	cdc2192	Language Guru
Edvard Eriksson	ehe2107	Manager
Nathan Burgess	nab2180	System Architect