YOLOP Language Reference Manual

Sasha McIntosh, Jonathan Liu & Lisa Li
sam2270, jl3516 and ll2768

1. Introduction

YOLOP (Your Octothorpean Language for Optical Processing) is an image manipulation language designed to aid developers in programming applications to process their images and simplify complicated tasks. This language reference manual details the specific lexical and naming conventions, syntax notation, declarations and statements, as well as the built-in functions and standard grammar of YOLOP.

2. Lexical Conventions

2.1 Comments

Comments begin with this symbols @# and terminate with the symbols #@. Only the text inside the symbols are commented out. There is no separate definition for single-line comments.

2.2 Variable Names

Variable Names are made up of a sequence of letters. Only the first 15 characters are significant.

2.2.1 Keywords

Keywords in the language and cannot be used in any way other than their respective purposes. These words are:

for	if	else	pixel
int	string	global	while
return	break	continue	function
print	read_in		

2.2.2. Types

integer	string	pixel

3. Syntax Notation

Tokens in a program are a series of characters broken up by separators. Spaces are considered whitespace and a series of spaces is treated as a single whitespace character. Tokens must be separated by a separator or an operator in order to be valid.

3.1.1 Token Separators

Token separators include whitespace characters and semicolons and are used to distinguish one token from another.

3.1.2 New Line Separators

A new line separator is used to distinguish the end on one line in the program from another.

3.1.3 Grouping Separators

Grouping separators are used to group sets of tokens. Grouping separators include parenthesis (), tab spaces, and brackets []. The correct usage of these separators is discussed in depth later in the manual.

4. Naming

4.1 Identifiers

Function and variable names are identifiers and can be made up of alphanumeric characters (0-9 and a-z) as well as underscores. The identifier must begin with either an underscore or a letter and is case sensitive. While a name can be infinitely long, only the first 15 characters will be used to reference an identifier.

Example: *function a_1:* *@# legal #@*
function 1_a: *@# illegal #@*
@# identifier cannot begin with a number #@

int _12345678901234444 *@# legal #@*
int _12345678901234555 *@# illegal #@*
@# variable name is already in use #@

5. Expressions

5.1 Primary Expressions

constant

An integer type constant is a primary expression.

string

A string is a primary expression.

pixel

A pixel is a primary expression.

primary-expression[]

Closed brackets following a primary expression indicate an array type of that primary-expression.

[expr]

Closed brackets around an integer number indicate access to an array element in the position determined by the contained integer.

(expr1 expr2 expr3)

Parentheses are used only in declaring the pixel primary-type *px* and its three values.

semicolons (;)

Semicolons are used in *for* loops and function calls to separate expressions.

colon (:)

Colons, used conjunctively with indentation, are used to indicate block statements.

comma (,)

Commas are used exclusively when writing multiple variable declarations that are of the same type.

5.2 Unary Operators

- expr

This unary operator groups right-to-left, and has a result of the negative of the expression, with the same type. The *expr* type must be an integer.

5.3 Mathematical Operators

expr + expr

The operator + groups left-to-right. It computes the sum of two expressions.

expr - expr

The operator - groups left-to-right. It computes the difference of two expressions.

expr * expr

The operator * groups left-to-right. It computes the product of two expressions.

expr / expr

The operator / groups left-to-right. It computes the quotient of two expressions.

expr++

The operator ++ groups left-to-right. It increments an expression by 1.

expr--

The operator ++ groups left-to-right. It decrements an expression by 1.

5.4 Relational/Equality operators

expr < expr

The operator < groups left-to-right. It returns 1 (meaning true) when the value of the left expression is less than that of the expression on the right, and 0 otherwise.

expr > expr

The operator > groups left-to-right. It returns 1 (meaning true) when the value of the left expression is greater than that of the expression on the right, and 0 otherwise.

expr == expr

The operator == groups left-to-right. It returns 1 (meaning true) when the left expression is exactly equal to the value than the expression on the right, and 0 otherwise.

expr ~= expr

The operator ~= groups left-to-right. It returns 1 (meaning true) when the left expression is not exactly equal to the value than the expression on the right, and returns 0 otherwise.

5.5 Logical Operators

expr1 && expr2

Evaluates to 1 when both expressions evaluate to non-zero values. There is a left-to-right evaluation. If *expr1* evaluates to 0, then *expr2* will not be evaluated.

expr || expr

Evaluates to 1 when either expression evaluates to a non-zero value. There is a left-to-right evaluation. If *expr1* evaluates to 1, then *expr2* will not be evaluated.

5.6 Assignment Operator

lvalue = expr

All assignment operators are right associative. They assign the evaluation of *expr* to the object

lvalue.

lvalue* =(op) *expr

This is equivalent to

lvalue = *lvalue* (op) *expr*

where (op) is an operator

6. Declarations

6.1 Variable Declarations

Variable declarations have the form *specifier var_name.* To declare multiple variables with the same type, simply separate each *var_name* with a space. Type specifiers are *int* for integers, *string* for strings and *px* for pixels. A variable cannot be created without a type specifier.

Examples:

```
int x = 6                    @# Integer Declaration  #@
string y = "why"             @# String Declaration  #@
px z = (255 255 255)  @# Pixel Declaration  #@
int[] x = [6 70]             @# Integer Array Declaration  #@
string[] y = ["why" "me"]    @# String Array Declaration  #@
px[] z = [(1 2 3) (1 2 3)]   @# Pixel Array Declaration  #@
```

6.2 Function Declarations

Function declarations have the form *function var_name [var_dec var_dec...].* The contents of the function include all lines immediately following the function declaration with one or more indents made with the tab key. The function parameters are contained within a set of brackets, separated by a space and in the form of a variable declaration.

Examples:

```
global int x, y, z = 6, 10, 4
function add x; y:
        return int z = x + y
        @# inside the function z = 16 #@
@# outside the function z = 4 #@
```

7. Statements

Statements are executed in sequence, and consist of either new object/variable declarations, modifying or assigning declarations, block statements, or control flow statements.

7.1 Conditional Statements

Conditional statements consist of either *if* or *if-else* blocks. As our language is based on indentation, this eliminates the ambiguity present in an *if expression if expression else* condition.

Examples:

```
if expression:
    if expression:
        statement
    else:
        statement
else:
    statement
```

7.2 Iteration Statements

Iteration statements consist of either *for* or *while* blocks.

7.2.1 For Loops

For loops must take in three expressions, separated by semicolons, and it evaluates *expression1* first and only once. *expression2* is a boolean expression and upon evaluating true, runs the *statement*. After the *statement* has been executed, *expression3* is executed. This continues until *expression2* evaluates false, at which point the for loop is terminated.

Example:

```
for expression1; expression2; expression 3:
    statement
```

7.2.1 While Loops

While loops take in one *expression*, which is a boolean expression that is evaluated each time before *statement* is executed. The while loop executes the *statement* repeatedly until the *expression* is evaluated as false.

Example:

```
while expression:
    statement
```

7.3 Jump Statements

Jump statements consist of *return*, *break*, or *continue* statements.

7.3.1 Return

The *return* keyword ends the function call by returning the value of an *expression* to the function's caller.

Example:

```
while expression:
    if expression:
        return expression
```

statement

7.3.2 Break

The *break* keyword terminates the smallest enclosing *while* or *for* loop, and jumps to the code displayed after that loop.

Example:
```
while expression:
    if expression:
        break
    statement
```

7.3.3 Continue

The *continue* statement allows control to pass to the end of the smallest enclosing *for* or *while* loop, essentially skipping over one iteration.

Example:
```
while expression:
    if expression:
        continue
    statement
```

7.3.4 Print Statement

The *print* keyword allows only a *string* to be printed into the console.

Example:
```
print string
```

8. Built-in Functions/System Functions

8.1 Input/Output (I/O)

8.1.1 Printing to a stdout

__print[String s] will print the string s to stdout

8.1.2 Reading from stdout

__scan[String s] will read the next line from stdout into the string s

8.1.3 Printing to an Image File

_printf[String s , String f] will print the string s to the file f.jpg

8.1.4 Reading from an Image File

_scanf[String f] will read from a file named f.jpg

9. Scope Rules

9.1.1 Global Scope

Global constants defined outside of any function calls are noted as global constants and contain a global scope. They can be accessed by any statements or expressions within the file and are noted by the preceding keyword *global*.

Example:
```
global int hello = 25
function add x; y:
    return x + y + hello      @# this is allowed because hello has #@
                              @# been defined as a global constant #@
```

9.1.2 Block Scope

Blocks contain a local scope, where variables defined within a block are local and accessible within that block. Blocks can access previously defined local variables in their parent's block as well as global variables.

Example:
```
function add x; y:            @# x and y are local variables #@
    int z = x + y             @# z is a local variable #@
    if x == 2:
        int a = x + 2  @# a and x are accessible local variables #@
        return a+2
    if x < 2:
        return a          @# error: a does not exist in this scope #@
    return z              @# z is within the local scope #@
```