Colode

Language Reference Manual

Steven Bonilla (sb3914@columbia.edu) - Manager

Tahmid Munat (tfm2109@columbia.edu) - Language Guru

Takuma Yasuda (ty2353@columbia.edu) - Language Guru

Willie Koomson (wvk2101@columbia.edu) - System Architect

Dimitri Borgers (djb2195@columbia.edu) - Tester

1. Introduction

1.1 Language Description

Colode is an efficient programming language used for manipulating individual or groups of pixel values. With a syntax that exploits the very best of both Python and Java, this language provides an extremely easy-to-learn and advanced way to modify image files for any use. The built-in functions can be used for the most common editing features available on other expert platforms. With the ability to control individual pixels on a loaded image file, a user will be able to alter anything at the most detailed level. In addition, by grouping pixels together, the process of editing will also become much more convenient and less time consuming.

1.2 Document Organization

In the following pages of this document, the syntax and core semantics of the Colode language are described. Chapter 2 outlines the types available to store data, subdivided into “primitive” and “built-in” categories. Chapter 3 will outline the lexical structure, which includes the low-level syntax that determines the basic rules of the language (what characters are used for comments, how statements are distinct from each other, etc). Chapter 4 delves into syntax notation, which specifies the precedence, expressions, and function calls permitted. Lastly, the final chapter lists the in-built functions provided within the Colode program.

2. Types

Types define how data can be stored and in what format, which allows the compiler to know how data will be used. Colode is a strictly typed language, so the type of each variable and return type of every function must be specified. Each type is defined by a variable name, variable type, and an attributed value (either during initialization or after). Names can be alphanumeric, are case sensitive, and may contain “_” to connect two or more words.

2.1 Primitive Data Types

Types	**Description**	**Examples**

int	4 byte signed integer type. Overflow not allowed. Values between -2,147,483,648 and 2,147,483,647.	`int x; x = 45;`
		`int y = 45;`

float	4-byte floating point number.	`float y;`
		`float x = 47.35;`

bool	8-bit boolean variable represented by true (1) or false (0).	`bool x = true;`
		`bool y = false;`

char	1-byte ASCII character.	`char x = ‘x’;`

list	Lists are an ordered, iterable collection of single-type data. Indexed using bracket notation.	`char list listName = [ ‘a’, ‘b’, ‘c’, ‘d’, ‘e’ ];`
		`listName[0]; // ‘a’`
		`listName[4]; // ‘e’`

string	Immutable null-terminated list of chars.	`string argName = “hello”;`
void	Function returns void when there is nothing to return.	`def void funcName():`

2.2 Built-in Data Types

Types	**Description**	**Examples**
Image	Images hold pixel values in a struct of 4 width x height matrices (1 for each channel). Pixel values can be accessed used image[width][height]). Images are initialized using the `coload` builtin function. Properties:	`Image img = coload(“omg.jpg”);`

	- *int* width - *int* height - *Matrix* red - *Matrix* blue - *Matrix* green - *Matrix* alpha	
Pixel	RGBA, 4-vector of floats Properties: - *float* r - *float* g - *float* b - *float a*	`Pixel p = img[0][0];`

Matrix	Matrices are a 2 dimensional data structure of fixed row length and column height (these are set implicitly at declaration). Values may be indexed (from 0) by their row and column integer, e.g. `mat[`*`row`*`][`*`column`*`]`	`Matrix m = [ 0 0 1 | 0 1 0 | 1 0 0];`
		`m[0][2] ; // 1`
		`m[1][0] ; // 0`

3. Lexical Conventions

3.1 Identifiers

Identifiers are used for variable declaration. Identifiers must begin with a lowercase letter, and can then be followed by any order of letters, numbers, and underscores (an identifier that tersely explains the data type is recommended).

3.2 Keywords

Keywords are reserved, and thus cannot be used as identifiers, because they have a special meaning to the Colode program. The following table demonstrates the specifications.

int	float	bool	string	false
true	return	continue	break	else
elif	if	while	for	def
void	in	range	Image	Pixel
Matrix				

3.3 Literals

Integer	A sequence of digits	`0, 3, 7162, 10`
Float	A number literal with an integer and fraction part. The integer part is written first, followed by a dot (.) followed by a fractional part. The decimal is required for the literal to be parsed as a float instead of integer.	`0., 3.1415`
Boolean	The keyword `true` refers to a truth value (1), while `false` refers to a false value (0)	`true, false`
Character	A single ASCII character surrounded by single quotes. The ‘ character must be backslash-escaped within character literals.	`‘A’, ‘!’, ‘\t’`
String	A sequence of ASCII characters surrounded by double quotes. The “ character must be backslash-escaped within string literals. Additional escape sequences are as follows: \n New line \r Carriage return \t Tab \\ Backslash	`“The quick brown fox jumped over the lazy dog”, “Hello World!\n”`
Matrix	Matrix literals are sequences of numeric literals surrounded by square braces. Each row is delimited by the pipe character. Row items are space-delimited.	`[0 1 | 0 1], [0 0 0], [2.5 |0.2]`

3.4 Comments

Comments are single-line only, and denoted by a double forward-slash ‘//’

```
    // This is a comment
    // This is another comment
```

3.5 Operators

Operator	**Description**	**Associativity**
Arithmetic	+, -, *, /, ^,%	Left to Right
Logical	and, or, not	Left to Right
Comparison	<, >, <=, >=, ==, !=	Left to Right
Assignment	=, +=, -=, *=, /=	Right to Left

3.5.1 Arithmetic

Arithmetic operators are used on the primitive data type of int and float. The resulting type is preserved. Currently supported operations are addition(+), subtraction(-), multiplication(*), division(/), raising-power(^), and mod(%).

For the additive operators, both operands are qualified versions of compatible object types. The result of the ‘+’ operator is the sum of the operands. The result of the ‘-’ operator is the difference of the operands.

Operands of ‘*’ and ‘/’ must have arithmetic type. The binary ‘*’ operator indicates multiplication, and its result is the product of the operands. The binary ‘/’ operator indicates division of the first operator (dividend) by the second (divisor). If the value of the divisor is 0, it would throw an error. The binary ‘%’ operator yields the remainder from the division of the first expression (dividend) by the second (divisor). The remainder has the same sign as the dividend, so that the following equality is true when the divisor is nonzero: *(dividend / divisor) * divisor + dividend % divisor == dividend*. If the value of the divisor is 0, it would throw and error.

Expression	**Result**	**Comments**
3 + 2 * 6	15	Multiplication is done before addition.
3 + (2 * 6)	15	Parentheses follow the precedence rules
(3 + 2) * 6	30	Parentheses override the precedence rules.

3.5.1 Logical

The *‘and’* and ‘*or’* operators associate left to right.The result has type int. For ‘and,’ if neither of the operands evaluates to 0, the result has a value of 1. Otherwise it has a value of 0. For ‘or,’ if either of the operands evaluates to one, the result has a value of 1. Otherwise it has a value of 0.

Expression	**Result**	**Comments**
true or true and false	1 (true)	Logical ‘and’ has higher priority than logical ‘or’
true or (true and false)	1 (true)	Parentheses follow the precedence rules
(true or true) and false	0 (false)	Parentheses override the precedence rules

3.5.1 Comparison

The relational(< <= > >=) and equality(== !=) comparison operators are of binary class with a left to right associativity. The operators <(less than), >(greater than), <=(less than or equal to), and >=(greater than or equal to) all yield a result of type int with the value 0 if the specified relation is false, and 1 if it is true.

The operands must be both arithmetic, in which case the usual arithmetic conversions are performed on them.

Expression	**Result**	**Comments**
4 + 1 > 2	1 (true)	Addition is done before comparison
(4 + 1) >= 2	1 (true)	Parentheses follow the precedence rules

The ==(equal to) and the != (not equal to) operators are exactly analogous to the relational operators, except for their lower precedence. For example, a < b == c < d is 1 whenever a < b and c < d have the same truth value.

Expression	**Result**	**Comments**
4 + 1 == 2 + 3	1 (true)	Addition is done before equality comparison
(4 + 1) == 2 + 3	1 (true)	Parentheses follow the precedence rules

3.5.1 Assignment

All assignment operators associate from right to left. The operands permissible in simple assignment(=) must be both arithmetic types or are of compatible structures. In the case of '=' the value of the right operand is converted to the type of the assignment expression, and replaces the value of the object referred to by the left operand.

For the operators += and -=, both operators must have arithmetic types.

3.5.5 Matrix operations

Colode includes support for a number of matrix operations. The arithmetic operations work on matrices, but transposition and convolution are also supported.

Transposition	@	A unary operator that returns the transpose of the operand matrix	`Matrix mat2 = @mat1;`

Convolution	**	A binary operator that returns the two-dimensional convolution of its two operand matrices.	`Matrix a = [1 1 1|0 0 0|1 0 1];`
			`Matrix b = a;`
			`Matrix c = a ** b;`

3.6 Punctuation

Each statement (besides statement blocks) are punctuated with a semicolon at the end to denote sequencing. Array literals use commas to separate items, and function calls use commas to separate function arguments. A pair of expressions separated by a comma is evaluated left to right.

```
int a = random(seed, param);
bool list b = [ true, false ];
```

4. Syntax Notation

Colode programs are a series of statements and declarations that are executed in top-to-bottom order.

4.1 Expressions

An expression is a combination of values, variables, operators, and functions to be evaluated. There are two types of expressions: one is assignment, and the other is typically arithmetic or boolean expressions.

4.1.1 Assignment

```
<identifier> [= <expression>];
```

Expression is evaluated and stored into the identifier.

Example:

```
x = 10
```

is an assignment expression which evaluates to 10.

4.1.2 Arithmetic Expressions

```
<expr_1> <op> <expr_2>;
```

Example:

```
2 + 3
```

is an arithmetic expression, which is composed of literals (2 and 3) and an arithmetic operator +. This evaluates to 5.

4.1.3 Boolean Expressions

```
<expr evaluates to true or false>;
```

Example:

```
2 != 5
```

is a boolean expression, which is composed of literals (2 and 5) and a comparison operator !=. This evaluates to true.

4.2 Declarations

4.2.1 Variable Declaration

The syntax for declaring variables is:

```
    <type> <identifier> [= <initial_val>];
```

Variables may be optionally initialized at their declaration. To declare the variable as a list type, the individual type must be written together with list, e.g. int list a;

```
    int i;
    bool b;
    string s = “Hello World”;
    string list buf = [“The”, “quick”, “brown”, “fox”]
```

4.2.2 Function Declaration

The syntax for function declaration is:

```
    def <identifier> ([<type> <param>, ...]) : <return type> {
```

```
		<function body>
	}
```
Functions may only be declared at the top level of a Colode file, not within another function. Functions that do not have a void return type must have a return statement that corresponds to the return type in the function declaration.

4.3 Initialization

Variables may be initialized at or after their declaration, by assigning an expression of corresponding type

```
	int i;
	i = 7;

	int n = 16;
	string s = “Hello World”;
```

4.4 Statements

A statement may be either an expression, a variable declaration, or one of the following control-flow statements. All statements that do not contain blocks must be ended with a semicolon.

4.4.1 if / elif / else

“if” is used for executing conditional statements. The syntax for an if statement is:

```
	if(<expr>){
		<suite>;
	}
```
expr must be a boolean expression to be evaluated, and suite is a sequence of statements. If expr is true, then the program goes into the brackets and executes suite. If expr is false, then the program ignores suite and proceeds to the next operation.

“if / elif / else” is used for multiple conditional branching; the syntax is:

```
	if(<expr_1>){
		<suite_1>;
	}
	elif(<expr_2>){
		<suite_2>;
	}
	elif(<expr_3>){
		<suite_3>;
	}
	......
	else{
		<suite_n>;
	}
```

The program sees each boolean expression one by one and executes a suite only when the corresponding expression is true. If nothing matches, then the suite in the else statement is executed. Boolean expressions, `expr_1` through `expr_n` *should be mutually exclusive* to ensure a logically correct control flow. If not, the programmer has to make sure that earlier expressions are caught in the order they are expected. See Example 2 below regarding this comment.

Example:

```
	int i = 5;
	if(i%3 == 1){
		print("i = 3k+1");
	}
	elif(i%3 == 2){
		print("i = 3k+2");
	}
	else{
		print("i = 3k");
	}
```

Example 2: ** *Notice how the first 2 expressions are both true! So the programmer has to specify the order of the expressions carefully. The control flow here is just like Java* **

```
	int i = 6;
	if(i%2 == 1){
		print("i = 3k+1");
	}
	elif(i%3 == 2){
		print("i = 3k+2");
	}
	else{
		print("i = 3k");
	}
```

4.4.2 Loops (for, while, continue, break)

"for" is used for iterative executions over a list, a string, or a certain range of integers. The syntax is as follows:

```
	for <variable> in <list>{
		<statement>;
	}

	for <variable> in <string>{
        <statement>;
}

for <variable> in range(<num>){
        <statement>;
}

for <variable> in range(<num_1>, <num_2>){
        <statement>;
}
```
For the first one, each element of the list is assigned to the variable. For the second one, each character that constitutes the string is assigned to the variable. For the rest, each integer in the specified range is assigned to the variable. The program executes the statement for one value in the list/string/range, and after that, move to the next value. For the third one, it starts from 0 and ends at num - 1. For the last one, it starts from num_1 and ends at num_2.

Example 1:
```
for x in [2, 3, 5, 7]{
        print(x);
}       //print “2 3 5 7” on screen
```

Example 2:
```
for str in “Hello World!”{
        print(str);
}       //print “H e l l o  W o r l d !” on screen
```

Example 3:
```
for x in range(4){
        print(x);
}       //print “0 1 2 3” on screen
```

Example 4:
```
for x in range(2, 5){
        print(x);
}       //print “2 3 4” on screen
```

“while” is also used for iterative execution. An iteration continues as long as the specified condition is true. The syntax is:
```
while(<condition>){
        <statement>;
}
```
condition must be a boolean expression.

Example:
```
      int i = 5;
      while(i > 0){
            print(“Hello”);
            i = i - 1;;
      }     //print “Hello” five times on screen
```

“continue” is used for ending current iteration of a for/while loop and starting with the next iteration, followed by a semicolon.

“break” is used for ending the iteration of the nearest enclosing for/while loop, followed by a semicolon.

4.4.3 Return

Return is used for ending the current function execution and return a value or multiples values
```
      def times(int a, int b) : int {
            return a * b;
      }
```

4.4.4 Statement Blocks

Statement blocks refer to a list of statements surrounded by curly braces that follow if/elif/else, for, while, as well as function declarations.

5. Standard Library Functions

The following are the standard library functions for the Colode programming language.

The Colode standard library provides functions for tasks such as image cloning, rotating, cropping, zooming etc. Currently, we do not handle memory management.

Function	**Description**	**Usage**
coload	Load any source image file for another available operation. Must be a valid image file of type .jpg or .png. Takes the image name(with the file extension) as the only parameter. Images must be in the same working directory.	`Image img = coload("image.png");`

coclose	Close a source image file. Takes an already initialized “Image” as the only parameter.	`coclose(img);`
coclone	Create identical copies from an initialized source image. (memory management is currently not handled). Takes an initialized “Image” as the only parameter.	`Image imgCopy = coclone(img);`
imgCopy	Allows Matrix manipulation to change the pixel values of an image or simply copy over an image by iterating through the Matrices. Takes 2 index values(int).	`imgCopy[imgCopy.width - s_width + i][imgCopy.height - s_height + j];`
print	Takes a String as the only parameter and prints it to the terminal. Int and long are converted to a string.	`print(“Hello World!”);` `print(10);`