Boa
Language Reference Manual
COMS W4115

Zack Whittaker
Zw2401@columbia.edu

June 27, 2016

Contents

1. Introduction

Boa is a simple language which allows for easy expression of mathematical and logical expressions. Its functionality and syntax is somewhat inspired by Python. This Language Reference Manual covers the syntax and semantics of Boa.

2 Lexical Conventions

2.1 Comments

Boa's comment system borrows the single hash '#' from Python [1] to indicate comments; however, comments in Boa are multiline comments which end with a mandatory second hash. A single symbol is used to ease the process of commenting and uncommenting sections of code and the hash sign is not allowed anywhere else in the syntax other than in strings

2.2 Identifiers

Identifiers in Boa must start with a letter either upper or lowercase. The remaining characters of the identifier can be any alphanumeric character or an underscore. Identifiers cannot be the same as any of the reserved keywords.

2.3 Reserved Keywords

The following keywords are reserved

if	int	and	do
else	float	or	print
end	string	xor	stringofint
function	return	while	

2.4 Variable Types

2.4.1 Integers

An integer in Boa is a string of digits 0-9. The default size of integers is 32 bits with one bit for the sign for a range of -2,147,483,648 to 2,147,483,647.

2.4.2 Floats

Floats are used in Boa to represent any value which is too large to be represented by a 32 bit integer or contain a decimal portion. Declarations of floats follow the conventions of the C language as described in [2] with a minor exception.

> A floating constant consists of an integer part, a decimal part, a fraction part, an e or E, an optionally signed integer exponent and an optional type suffix, one of f, F, l, or L. 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. The type is determined by the suffix; F or f makes it float, L or l makes it long double, otherwise it is double.

In Boa, there are no long or double values so there is no suffix on the initialization of a float; including one will cause the lexer to throw an error. Floats are also 32 bits of which one bit is for the sign of the number portion, 24 bits are used for the precision of the float, and the remaining 8 bits are used for the exponential portion of which the first bit determines the sign of the exponent. This allows for values of the exponent between +/- 127.

2.4.3 Strings

A string in Boa is any string of characters which is surrounded by quotations “”. As with Python, the escape character for Boa is the backslash character “\” [3]. This is used to add newlines **\n**, quotes **\”** or even a backslash \\

2.4.4 Arrays

Arrays in Boa are a way to refer to a list of data by a collective name. Arrays can only represent data in one dimension and all elements of the array must be of the same type.

2.5. Operators

There are many operators in Boa which cannot be considered keywords as they contain character which are not allowed in identifiers. These are also lexical conventions, but not considered keywords as they are not legal names for identifiers.

2.6 Delimiters

Whitespace characters (space, tab and newline) are used to delimit tokens in Boa.

3 Syntax

3.1 Lines

Boa scripts are made up of logical lines which are separated by a semicolon “;”. New line characters are used only to delimit tokens and make the code more readable. Lines are made of expressions.

3.2 Blocks

Although Boa is inspired by Python, the lexer for Boa does not use indentation as a means to organize code. Blocks in Boa take the place of indentation in Python and are used to combine multiple lines so that they may be referred to collectively. Blocks are indicated by curly braces "{" and "}" and can be nested indefinitely. Blocks may also contain no statements at all.

3.3 Functions

User-generated functions must be declared before they are called. The syntax to create a function in Boa is as follows:

function *identifier (parameters) = {block}*

where *block* must contain a return statement. Within *parameters*, the type of each parameter expected must proceed the name of the formal parameter which will be used in the body of the function. There can be multiple parameters passed into a function of different types with each separated by a comma.

3.3.1 Return

return(expr)

The return statement is a requirement of functions in Boa. Only one variable may be returned; however, as will be discussed in section 5, variables whose scope includes the function may be altered from within the function. Since a function call is an example of an expression, it is not possible to return nothing from a function. The user must return something, but this thing does not have to be of importance if the function call is not used within a larger expression.

3.4 If-Then-Else Statements

If-then-else is a statement in that it does not evaluate to anything as is the case with expressions to be discussed later. Instead, if-then-else is used as a control structure to control which statements and expressions get evaluated during run-time. This structure is formatted as follows

if (logical expression) *then* {block to evaluate if true} *else* {Block to evaluate if false};

The *else* is required in Boa as are the curly braces. Both are used to avoid ambiguity and improve readability of the script.

3.5 Loops

While loops are the only loop which is used in Boa. This loop is another type of statement which does not return anything. The syntax for a while loop is as follows:

while (logical expression) *do* {Block to be executed if logical expression is true};

While the logical expression in the first block is true, the statements in the block following *Do* will be executed.

3.6 Expressions

An expression in Boa is a statement which can be evaluated and yields one of the primitive types in boa described in 3.6.2. Expressions encompass nearly every construct in boa with the exception of if-then statements and loops.

3.6.1 Literals

Literals are the simplest of all expressions in Boa. These are integers, floats, or strings which are typed directly into the script and "return" their literal value. String literals must be offset by quotes. Array literals are offset by square brackets and their elements separated by commas.

3.6.2 Variable Declaration

Variables must be declared before they are used. The variable type must precede the identifier in a declaration expression since assignment of the variable in this expression is optional. Variable types must be in lower case otherwise they will be recognized as identifiers. Therefore declarations for each of the variable types in Boa would be written as such:

```
int this_is_an_integer;              #initializes to 0

float this_is_a_float;               #initializes to 0.0

string this_is_a_string;             #initializes to ""

[variable type] this_is_an_array;    #initializes to an empty array
```

As shown in the comments above, upon creation, variables are automatically assigned to default values. These expressions add symbols to the symbol table of the block they are in which can be accessed in that block and all blocks which reside in that block.

3.6.3 Assignments

identifier = expr;

An assignment is an operator sets the value of a variable. These can be included in the initialization expression or in a later expression. For all but the array, a single equal sign is used for assignment to separate the identifier from the expression which determines its value as in the following assignments for the variables declared above.

```
this_is_an_integer = 42;

this_is_a_float = 2.7182818e-6;

this_is_a_string = "Four score and seven years ago";
```

In the case of the integer, if a float is on the right side of the operator, it will be converted to an integer by rounding down to the nearest integer. In the case of the float, if an integer is on the right side of the expression it will be converted to a float with a possible loss of precision. Strings assignments can only except strings on the right side of the assignment.

Arrays are somewhat different in that the left side of the assignment has added syntax to denote the location to which the value will be inserted. Array indexes are denoted by parentheses immediately following the array identifier. Indexes must be integers greater than or equal to 0. Consider the following assignment to an array of floats.

```
[float] array_of_floats;

array_of_floats(2) = 3.14159;

#Array of floats now has the value [0.0, 0.0, 3.14159]
```

As shown above, when the referenced index is greater than the length of the array, the array will be extended to that size to accommodate and all newly created indexes will be initialized to the default value for that type. This is only for the case of assignments. Referring to an index which is out of bounds in any other situation will result in an error.

Assignments return the value which is assigned. This allows them to be used as expressions with the side effect of changing the value in the symbol table. Lastly, in all assignments, the right operand can

3.6.4 Identifiers

After they have been declared, identifiers are valid expressions in Boa as they return the last value which was assigned to them in the current scope.

3.6.5 Array Access

Accessed array values are used in the same place as identifiers. To access an array value the identifier of the array is followed by an index in parentheses.

3.6.6 Parenthetical Expressions

Putting parenthesis around an expression prevents the parser from splitting up that expression when used as an operand for another expression.

3.6.7 Logical Expressions

The constructs for *if* and *while* require logical expressions to function correctly. Although Boa does not have Boolean variables, any expression which evaluates to a non-zero integer will be taken to mean true with zero values representing false.

3.7 Operators

Another way to form an expression is with operators. An expression can be made up of an operator which takes other expressions as its operands. These can be either unary expressions which act on a single operator or binary expressions which act on two operands to create another expression.

3.7.1 Unary operators

Unary operators act on only one expression and always precede the expression which they act on.

3.7.1.2 Negation

expr → - expr

Negation of an expression yields the additive inverse of that expression's evaluation. Negation can only be applied to integers and floats. For all other data types, the operation will result in an error.

3.7.1.3 Logical Negation

expr → !expr

Logical negation evaluates to a 1 if the expression which follows evaluates is non-zero and evaluates to 0 if the expression following the negation evaluates to some non-zero value. Logical negation has the same type restrictions as negation.

3.7.2 Binary operators

Expressions generated from binary operators use infix notation. Each has a level of precedence and a direction of associativity to remove ambiguities for the parser.

3.7.2.1 Addition

expr → expr + expr

This operator adds two variables. This operator is only valid when both operand expressions are either of type int or float. When both are of the same type, the expression will produce that type. When one is an integer and the other is a float, the integer will be converted with a possible loss of a precision and a float will be returned.

3.7.2.2 Subtraction

expr → expr - expr

This operator subtracts the evaluation of the right expression from that of the left expression. The same type restrictions apply to subtraction as with addition.

3.7.2.3 Multiplication

expr → expr * expr

This operator multiples the evaluations of the left and right expressions. The same type restrictions apply to multiplication as with addition.

3.7.2.4 Division

expr → expr / expr

This operator divides the evaluation of the left expression by that of the right expression. Division will return a float regardless of whether either operand is an integer.

3.7.2.5 Modular Division

expr → expr % expr

This operator acts the same as division above, however instead of returning a float, this operator returns the decimal evaluation rounded down to the nearest integer. This should be used when an integer is required as the return type from a division.

3.7.2.6 and

expr → expr **and** expr

and is a logical expression which evaluates to a if both the left and right expressions are non-zero integers and 0 of either operand evaluates to an integer with a value of zero. Any other type of operand will result in an error.

3.7.2.7 or

expr → expr **or** expr

or is a logical expression which evaluates to 1 if either of the operands is a non-zero integer and 0 if both expressions are integers with values of zero. **or** has the same type restrictions as **and**.

3.7.2.8 xor

expr → expr x**or** expr

xor is a logical expression which evaluates to 1 if exactly one operand is a non-zero integer and evaluates to 0 in all other cases. “xor” has the same type restrictions as “and”.

3.7.2.9 equality

expr → expr == expr

Equality is a logical operator which evaluates to a 1 is the two operands evaluate to the same value and 0 if they do not. In the case of integers and floats, the two expressions must evaluate to the same numerical value. In the case of strings, the equality operator checks if the contents of the strings are identical. The **equality** operator cannot be used on arrays.

3.7.2.10 Inequality

expr → expr != expr

Inequality is a logical operator which has the opposite evaluation as equality. That is, it is the logical negation of equality. Therefore, it has the same type restrictions as equality.

3.7.2.11 Greater than

expr → expr > expr

Greater than is a logical operator which evaluates to 1 if the evaluation of the left expression is greater than that of the right expression and evaluates to 0 if the evaluation of the left expression is less than or equal to that of the right expression. The operands can be of type float or int interchangeably. **Greater than** cannot be used to compare types of string or array.

3.7.2.12 Less than

expr → expr < expr

Less than is a logical operator which evaluates to 1 if the evaluation of the left expression is less than that of the right expression and evaluates to 0 if the evaluation of the left expression is greater than or equal to that of the right expression. The operands can be of type float or int interchangeably. **Less than** cannot be used to compare types of string or array.

3.7.2.11 Greater than or Equal to

expr → expr >= expr

Greater than or equal to is a logical expression which evaluates to 1 if the evaluation of the left expression is greater than or equal to that of the right expression and evaluates to 0 otherwise. **Greater than or equal to** has the same type restrictions as greater than.

3.7.2.12 Less than or Equal to

expr → expr <= expr

Less than or equal to is a logical expression which evaluates to 1 if the evaluation of the left expression is less than or equal to that of the right expression and evaluates to 0 otherwise. **Less than or equal to** has the same type restrictions as less than.

3.7.2.13 Concatenation

expr → expr :: expr

Concatenation is used to add the evaluations of two expressions side by side. Concatenation can only be used only on strings.

3.7.3 Function Calls

expr → function_call(parameters)

Any non-void function can be used as an expression which evaluates to value which is returned form the function.

3.7.4 Operator Prcedence

The relative precedence and the associativity of these operators is given below.

Level of Precedence	Operators	Description	Associativity
1 (Highest)	(expr)	parenthetical expressions	N/A
2	-x, !	negation, logical negation	N/A
3	*, / , %	multiplication, division, modular division	Left
4	+, -, ::	addition, subtraction, concatenation	Left
5	and, or, xor	logical arithmetic	Left
6	==, !=, >, >=, <, <=	logical comparisons	Left
7	=	Assignment	Right

4. Built-in Functions

Boa has very few basic functions. Many of these are necessary for printing and debugging.

4.1 Print

print (expr)

Print will print the string representation of the evaluation of expr. If this is an int or a float, the float will first be converted to a string. Arrays cannot be printed using the print function.

4.1.1 stringofnum

stringofnum(num)

stringofnum will accept either an int or float and return a string representation of that value. This is useful for printing out numbers.

5. Scope

Variable declarations and assignments in Boa are relevant only in the block which they occur and in all blocks which are children. When an identifier is used in an expression, it will have the value of the last assignment of that identifier in the current scope. Within functions, the variables will be referred to their given identifier within that function. Even if the passed parameter and that used in the function have the same name, changes to these variable from within the function will not be carried out of the function unless they are returned. Changes to variables which are not passed into the function or given a local definition will affect those variables which were found in a parent environment.

6. References

[1] Python Software Foundation. Python Language Reference, version 3.5. Available at http://www.python.org

[2] B. W. Kernighan. The C Programming Language. Prentice Hall Professional Technical Reference, 2nd edition, 1988.