HavaBol Interpreter Specification

OGP-b1.0

Authors:

Charles Dix

Elijah Arnold

Jake Willis

Table of Contents:

Section I: Introduction

This document describes the implementation of the HavaBol programming language, *OGP-b1.0*. It describes the HavaBol language, the functionality supported by this interpreter, and decisions made during implementation when they deviate from the distributed core requirements or were left up to the implementors. It also contains two appendices, one with functional test code and the second with various error cases and example output.

Decisions:

These sections will describe the decisions made and why.

What is HavaBol?.
HavaBol is an interpreted, imperative, statically-typed language that uses infix notation. It provides mechanisms for declaring primitive types including Ints, Floats, Bools, and Strings, as well as homogenous arrays consisting of these types.

Decisions:

This implementation of HavaBol uses infix notation, that is operators act as separators between operands rather than preceding the operands they apply to.

Example:

```
// A prefix expression adding two values, 2 and 3, and
// storing the result in a variable, x.
let x (+ 2 3)
```

In the above example, the first operation we encounter is indicated by the *let* keyword which indicates the next token is a variable, in this case *x*, which will be followed by an expression evaluating to a value we wish to store in it. That value is an infix expression where + is the operator and 2 and 3 are the operands. The equivalent HavaBol code is below.

Example:

```
// The equivalent HavaBol code in which infix notation is
// used.
x = 2 + 3;
```

> Here the first operand, *x*, precedes its operator, =, which is followed by the expression. This time the expression is an infix expression, 2 precedes + which precedes its second operand, 3.

HavaBol supports complex expressions containing multiple primitive types and includes a system for coercing between these, allowing expressions consisting of Floats, Ints, and even Strings if their format matches the HavaBol specification for Int and Float formatting. Expressions may also include array and String subscripts including support for nested expressions and subscripts.

Logical constructs provided by the HavaBol specification include if/else, while loops, and for loops with functionality for for loops allowing iteration via primitive integer constraints which can be derived from expressions, character iteration through Strings, and iteration by element through arrays.

Section 2: Types, Declarations, Initialization

This section details the primitive types supported by HavaBol, how to declare variables of these types, and proper initialization.

Section 2.1: Types

HavaBol supports four primitive types:

Int consists of a 4 byte integer values ranging from -2147483648 to 2147483647

Examples:

```
-8, 9, 33, “33”
```

Float consists of an 8 byte floating point value

Examples:

```
2.1, -24.18323, 3.14, “7.2”
```

Bool a boolean type where T represents true and F represents false

Examples:

```
T, F
```

String a type which consists of an ordered collection of characters, is mutable and can be of variable length including an empty String, these are marked by surrounding single or double quotes (“ or ‘)

Examples:

```
“foo”, ‘bar’, “”, “\ttabbed text”
```

Notes:

1. “” represents an empty String.
2. The last example contains an example of a String with an escape sequence, in this case \t representing a tabbed space.

Section 2.2: Declarations

Declaring variables allows the programmer to store, reference, and modify values, allowing them to be included in complex expressions, assigned to other variables, including arrays, or to be used in functions. Variables must have a declared type and are identified by a unique identifying token.

Decisions:

Masking, the reusing of variable names in later declarations, is not allowed by this HavaBol implementation, even in local inner scopes. This is done to reduce bugs related to masking and force programmers to use more relevant variable names.

Variable names must begin with an alphabetic character and may contain numbers or special characters as long as they are not HavaBol defined operators and separators. The names must also not be reserved HavaBol keywords. Operators and separators in variables names will be treated as separate tokens and may result in parse errors on execution.

Examples:

```
Int myval;
Float float42_32;
String S_t_r;
Bool foo;
```

Section 2.3: Assignments and Initialization

Giving values to variables is simple. The operator associated with this is '=';

Examples:

```
Int myval;  // declared variable with no value
myval = 3;  // myval now stores the integer 3
```

Variables are mutable and once assigned a value, may be changed at any point. Arrays, however, retain some key identifying data which will be discussed later.

Examples:

```
String helloStr;
helloStr = “hello”;        // first value assigned
helloStr = “Hello world!”; // THe value stored is changed.
```

It is, however, possible to assign values when variables are first declared by combining the notation for declaring and the notation for assignment.

Examples:

```
Int myInt = 3;
String foo = “foo”;
```

HavaBol also supports four combined assignment operators. These include the following: +=, -=, *=, and /=. These perform the operation indicated by the left symbol and use the left variable’s value as the first operand and the result of the expression to the right of the assignment operator as the second and store the result in the left side variable.

Examples:

```
Int foo = 3;
foo *= 2;      // Multiply the value of foo by 2 and store it in
               // foo.
print(foo);    // This will print 6.
```

Section 2.4: Type Coercion

While programming, it may be necessary to convert one type to another, perhaps an Int, Float, or Bool to a String. The programmer may for sake of flexibility wish to use the same assignment value for multiple variables of varying types. HavaBol is capable of coercing these values through assignments and expressions, the latter being covered in detail later. The following type coercions are allowed in this implementation.

Int → *Float, String*

Float → *Int, String*

Note: Int values produced from floats will have everything following the decimal truncated.

Bool → *String*

Note: A Bool with the value T yields a String with the value “T” and a Bool with the value F yields a String with the value “F”

String → *Float, Int*

Note: Strings being coerced in this way must match the formatting required for the respective types or a runtime error will occur.

Examples:

```
String intStr = “4”;
Int myVal = 2 * intStr;     // intStr can be coerced.
print(myVal);               // Prints 8.
```

Coercion is supported in any type of operation including complex expressions, basic and operator assignments, array assignments, and subscripts. Although we will discuss arrays later an array of Strings each representing Int or Float values could easily be converted to an array of those types by simple assignment or have its values used in expressions or assignments to primitive variables making this a powerful feature.

Section 3: Expressions

HavaBol supports various operators on its types. These include simple mathematical operations such as +, -, /, etc, boolean operations which yield HavaBol Bool types such as <, >, etc. and some operations on Strings such as concatenation. This section discusses the operators supported by this HavaBol implementation and order of operations used in evaluation of complex expressions.

Section 3.1 Operators

Numerical Operators	Boolean Operators	String Operators
+	==	#
-	!=	
*	<, <=	
/	>, >=	
^	and	
	or	

Listed above are the operators used in HavaBol. The four common numerical operations are performed using +, -, *, and /, as well as ^, which is used for exponentiation. For boolean operations, == indicates a comparison of equality between operands; please do not confuse the assignment operator, =, with the boolean comparison operator, ==. != tests if two operands are not equal. < stands for "less than," and <= stands for "less than or equal to." Similarly, > stands for "greater than," and >= stands for "greater than or equal to." **and** is for boolean logic "and" operations, as **or** is for "or" operations. The string operator, #, is for concatenating, that is adding, strings together. The operand on the right hand side of the # is appended to the end of the string on the left.

HavaBol has left-hand coercion for expressions, meaning the end result of operations depends on the datatype of the leftmost operand. Therefore, an Int operand plus a Float operand would result in an Int result, but a Float plus an Int would result in a Float. The exception to this is if when performing a numeric operation, then the result is coerced to a numeric even if the lefthand operand is a String.

Example:
```
Int i = 1;
Float j = 2;
Str k = “4”;
print("Int + Float =", i + j);
print("Float + Int =", j + i);
print(“String + Int =”, k + i);
```

Result:
```
Int + Float = 3
Float + Int = 3.0
String + Int = 5;
```

As stated above HavaBol's coercion rules allow for interesting operations. For example, an Int plus a String containing a number, will result in an Int that is the sum of the two values. A String concatenated with an Int would result in a String with the Int appended to the end. However, because operations depend on the datatype of the left hand operand, you could not concatenate an Int with a String. See the following example.

Example:
```
Int integer = 25;
String str = "25";
print("integer", integer, "+ str", str, "=", integer + str);
print("str", str, "# integer", integer, "=", str # integer);
print("integer", integer, "# str", str, "=", integer # str);
```

Result:
```
integer 25 + str 25 = 50
str 25 # integer 25 = 2525
havabol.ImproperOperationException: # is not a valid numeric
operation
```

By extension, we know that a numeric operation is only valid if there is a numeric, an Int or Float, on the left hand side of the numeric operator, a Bool for boolean operations, and a String for string operations.

As a final note, be aware that coercion of data types may yield results you do not expect if you do not carefully think expressions through. For example, 2 < 10 and 2 < "10" both yield a comparison between an Int and an Int; in the latter case, the String is coerced to take the value of an Int. However, "2" < 10 will coerce the Int on the right hand side to a String, leading to a lexicographic comparison of the ASCII values of characters within a String. 2 < 10 is true, but "2" < 10 is false.

Section 3.2: Order of Operations

Below is the order of operations for HavaBol operators from first, starting at 1, to last. u- indicates a unary minus, that is the operator for indicating a negative number.

Precedence of Operation	Operator
1	u-
2	^
3	* /
4	+ -
5	#
6	< > <= >= == !=
7	not
8	and or

Expressions within parentheses, (and), or brackets, [and], are evaluated before any of these operators are applied. Numerical operations follow typical PEMDAS rules; for those unfamiliar with the mnemonic, PEMDAS stands for Parentheses, Exponents, Multiplication, Division, Addition, and Subtraction, the order of mathematical operations.

Although the above order of operations may seem fairly obvious, it's important to note that boolean operations are performed last. This ensures that comparisons are between the results of expressions on either side of a boolean operator are evaluated, instead of the operands to their immediate left and right. The following example demonstrates this.

Example:
```
     Int i = 1;
     Float j = 2;
     print(i, "+", j, "=", i + j);
     print("(", j, "^ 4 ) /", j, "=", (j ^ 4) / j);
     if i + j < (j ^ 4) / j:
          print(i, "+", j, "< (", j, "^ 4 ) /", j, "is true");
     else:
          print(i, "+", j, "< (", j, "^ 4 ) /", j, "is false");
     endif;
```

Result:
```
     1 + 2 = 3
     ( 2 ^ 4 ) / 2 = 8.0
     1 + 2 < ( 2 ^ 4 ) / 2 is true
```

Please keep in mind that in HavaBol, unlike in some languages, boolean expressions do not short circuit. A boolean expression evaluated to false before an **and** will still process the operand or expression after the **and**; the same goes for a boolean expression evaluating to true before an **or**.

Section 4: Array Declarations, Assignments, and HavaBol Array and String Subscripting

In HavaBol, an array is a homogenous, ordered collection of primitives of a fixed size. Let's break this down to determine exactly what it means.

"...homogenous…"
A HavaBol array may only contain values of its declared type. An array of Ints may only contain Ints and an array of Bools may only contain Bools. HavaBol type coercion is used when assigning and referencing values though, allowing for some flexibility.

"...ordered collection…"
Values assigned to an array have an order. They are referenced using zero-based indexing in the order they are assigned. The "for each" mechanism HavaBol provides will iterate through the elements in an array in their assigned order.

"...of primitives…"
HavaBol arrays may only contain one of the four established primitives, not other arrays.

"...of fixed size…"
Arrays in HavaBol are bounded, that is once a size is declared or interpreted by assignment, the size of an array may not change, and any attempt to add more values to the array than the size allows for or attempts to reference or assign values outside of this bound results in an error.

Decision:

> HavaBol arrays in this implementation are bounded. Use of the *unbounded* keyword results in undefined behavior and referencing, assigning, or attempting to iterate through arrays that are not declared with or interpreted to have a fixed size will result in an error.

Now that we know a little about HavaBol arrays, let's take a closer look at how we can use them.

Section 4.1: Array Declarations

Declaring an array is similar to how primitive variables are declared with open and closed brackets being used to express that the value is an array.

Examples:

```
Int intArray[];
String strArray[4];
Bool boolArray[] = T, F, T, T, F;
```

Notice that on line 1 and 3 from the above example arrays in HavaBol do not need to have a declared size as is done on line 2. Size may be interpreted from assignment as is done on line 3. The size of the array *boolArray* in this example will be set to five and cannot be changed. Likewise if the following line of code were added:

```
    intArray = 3, 2, 1;
```

intArray's size would be set to three following that assignment and would remain fixed.

Section 4.2: Array Assignments

HavaBol supports several ways to assign values to arrays. The first is to list all values one wishes to add to an array.

Examples:

```
    Int intArr[] = 5, 8, "3", 8; // Remember, coercion allows "3"
                                 // to be a valid assignment.
    String strArr[4];
    strArray = "This", "is", "an", "array!";
    Bool boolArr[5] = T, , F, T; // Not all values are filled.
                                 // Although allowed, any attempt to
                                 // reference unitialized values
                                 // will result in a runtime error.
                                 // Be careful!
```

It is also possible to assign values from one array to another. In the cases where the arrays differ in size, all possible values will be copied either stopping when the left side array is filled or when there are no values in the right side array.

Example:

```
    Int arr1[] = 1, 2, 3, 4, 5;
    Int arr2[3] = arr1;         // arr2 now contains: 1, 2, 3
    Int arr3[] = -1, -2, -3;
    arr1 = arr3;                // arr1 now contains: -1, -2, -3
                                // Because its max size may not
                                // change after its size is
                                // interpreted, more values may be
                                // assigned to index three and four.
```

It’s also possible to make scalar assignments to arrays. In this case an array must have a declared or interpreted size prior to assignment.

Examples:

```
String helloArr[5] = “Hello World!”;   // All values of helloArr
                                       // are “Hello World!”;

Int intArry[];
intArr = 3, 2;                         // Size was not yet
                                       // declared or
                                       // interpreted so this is
                                       // an error.
```

Section 4.3: Subscripting

With values assigned to arrays knowing how to reference those values makes them useful. In Havabol it’s as simple as referencing their position index from 0.

Example:

```
Bool boolArry[] = T, F;
print(boolArray[0]);    // This prints T!
```

Elements referenced this way may be used in expressions and in assignments, both to primitive variables and as scalar assignments to other arrays.

Example:

```
Int intArr[] = 3, 2, 1;
int myInt = 2 * intArr[1];      // Reference the value 2 at
                                // index 1.
print(myInt);                   // Prints 4.
Int allFours[4] = 2 * intArr[1]; // As above, we reference the
                                // value 2 and the result of
                                // this expression, 4, is set to
                                // all four values of
                                 // the allFours array.
```

Finally, subscripts in arrays may be negative. This allows the programmer to reference values from the end of the array. -1 represents the last element of an array and the range of values allowed for referencing an array using negative subscripts ranges from -1 * [*array size*] to -1 with the former being equivalent to the zeroth index.

Examples:

```
     Int myVals[] = 1, 2, 3, 4;
     Int val = myVals[-1];      // val equals 4;
     val = myVals[-3];          // val equals 2;
     val = myVals[-5];          // This is an error as -5 < -1 * 4
                                // where 4 is the interpreted length
                                // of the array from its
                                // initialization on line 1.
```

You may have noticed that Strings are mentioned in the section 4 heading but why? Although Strings are treated as HavaBol primitives, as stated before they support their own subscripting system similar to arrays. This means we can reference and set values of Strings using subscripts and even perform some de facto operations this way, but what does this look like?

As with arrays, left and right side brackets are required for using String subscripts, and as before you may even use negative subscripting. However, the programmer should be aware that referencing a String subscript outside of the the String's range, either negative or positive, results in an error.

Examples:

```
     String helloStr = “hello world!”;
     String helloChar = helloStr[1];
     print(helloChar);                  // Prints e!
```

We can even set Strings using subscripts. This will override the element at the index with the String we are assigning. If the String being assigned extends past the last index, the value it is assigned to will be extended and its remaining values overridden.

Examples:

```
     String myStr = “brace”;
     String myStr2 = “elet’:
     myString[4] = myStr2;      // myString[-1] could also be used.
     print(myStr);              // Prints “bracelet”!
```

Section 5: If Statements

HavaBol provides several logical constructs for manipulating flow of control to execute code conditionally. The first of which we'll discuss is "if statements".

Decision:

In writing this interpreter we enforce a strict static local scope. Any variables declared inside of control flow constructs such as if statements, while, and for loops are removed from the symbol table and storage after flow of control returns to the containing block of code or after each iteration in the case of loops. Attempting to reference these variables will result in an error as if they were never declared and initialized.

If statements specify the conditional execution of two branches according to the value of a boolean expression and follow the following format.

```
if condition:
   statements;
endif;
```

or:

```
if condition:
   statements;
else:
   statements;
endif;
```

statements represents the block of code to be executed conditionally. If *condition* evaluates to true, T in HavaBol, *statements* is executed otherwise if condition evaluates to false, F in HavaBol, *statements* is not executed or the else branch is executed. *condition* can be any expression as long as it evaluates to a Bool. If multiple if statements are nested, then the else is always associated with the closest preceding if statement.

The keywords if, else, and endif should be used as depicted below.

Examples

```
Bool foo = T;
if foo:
    print ("hello world");
endif;
// The above prints “hello world” as foo evaluates to a
// HavaBol Bool with the value T.

Int x;
x = 5;
if x > 10:
    print("x is greater than 10");
else:
    print ("x is low");
endif;
// Prints “x is low” as the operation “>” in the condition
// expression evaluates a HavaBol Bool with the value F.
```

Section 6: For Loops

The second logical construct we'll discuss is for loops. For loops allow us to execute commands for a number of iterations. HavaBol supports three types for loops:

limit-by: initialize a numeric value and increment by a set value until the limit
elem-in: iterate through each element in an array
char-in: iterate through each character in a String

Section 6.1: Limit-by For Loops

Limit-by loops allows the execution of commands for a number of iterations and follow the following format.

```
for cv = sv to limit by incr:
    statements
endfor;
```

The values *cv*, *sv*, *limit*, and *incr* are placeholder values in the above format with *for*, *to*, *by*, and *endfor;* being the essential keywords.

The 'for' loop executes a sequence of commands for certain number of iterations determined by the control variable (*cv*), incrementer (*incr*), and *limit*. Once *cv* is larger than or equal to the *limit* the loop will stop executing. The *incr* and *limit* are only evaluated once upon entering the loop. Once the loop has started there is no way to update the values of the *incr* and *limit*. The control variable on the other hand can be updated once the loop has started. if there is no specified *incr* the default value is one.

Examples:

```
Int iCM[10] = 10, 20, 30, 40, 50, 60;
inc = 1;
n = 6;
for i = 0 to n by inc:
    print("\t", iCM[i]);
endfor;

Int i;
Int n=4;
Float data [10] = 90.0, 50.0, 60.0, 85.0;
for i=0 to n:
    print ("i=", i, data[i]);
    n += 1;
endfor;
```

HavaBol does not explicitly require that a control variable is declared, however, once a loop completes execution, the control variable remains valid. This means that it cannot be declared again, since HavaBol does not allow masking.

Section 6.2: Elem-in For Loops

The second for loop type supported by HavaBol allows for easy iteration through array elements and follows the following format.

```
for item in array:
    statements
endfor;
```

As before the keywords *for* and *endfor;* are present and the new keyword *in* is introduced. *item* and *array* are placeholders with *array* being a valid, declared, and initialized array variable and *item* being any desired variable name as long as it matches the HavaBol requirements.

For each element in the array, the code within the loop represented by *statements* will be executed and the variable used in place of *item* will hold the value of the current element in the array for that iteration.

As mentioned before, elements in an array can be left uninitialized. The elem-in for loop will iterate through all values in an array up to the last declared element, however, encountering an uninitialized element prior to the last declared element will result in a runtime error.

Modifying an array inside the for loop may not alter the amount of loop iterations and attempting to reduce the number of elements in an array will result in a runtime error.

Upon exiting the loop, the variable indicated by *item* in the above format will remain valid and attempting to declare it again breaks HavaBol's rule against masking, resulting in a runtime error.

Examples:

```
Int iCM[10] = 10, 20, 30, 40, 50, 60;
for iVal in iCM:
    print("\t", iVal);
endfor;

Int myVals[] = 1, 2, 3, ,5;      // This is a valid declaration
                                 // but contains uninitialized
                                 // values.
for val in myVals:               // An error will occur on the
     print(val);                 // third iteration when an
endfor;                          // uninitialized value is
                                 // encountered.
```

Section 6.3: Char-in For Loops

The last supported for loop type in this HavaBol implementation is the char-in loop. This allows for simple iteration through HavaBol Strings character by character and follows the following format.

```
for character in string:
        statements
endfor;
```

Again, the *for* and *endfor* tokens appear, and similar to the elem-in loop, the *in* token is required. However, *string* in the above format example represents a HavaBol String type, not an array, and *character* will be a variable name decided by the programmer whose value changes each iteration to store the String representation of the current character in the String.

As with arrays, modifications to the *string* variable may not modify execution of the loop and attempting to do so will result in a runtime error.

The variable represented by *character* will remain valid and as before cannot be redeclared within the current local scope.

Examples:

```
String sleep = 'Snooz';
for ch in sleep:
    print ('ch=', ch);
endfor;
```

Section 7: While Loops

The last control flow construct we'll discuss is while loops. Like if statements, while loops allow simple execution of code based on an expression which must evaluate to a Bool value. While loops execute the code they encompass until the expression evaluates to a Bool F and follow the following format.

```
while cond:
    statements
endwhile;
```

The necessary keywords are *while* and *endwhile* while *cond* represents any complex or simple expression which evaluates to a Bool.

Examples:

```
Int condInt = 0;
while condInt < 5:
    condInt += 1;
    print(condInt);
endwhile;
print("While ended when condInt equalled", condInt);
```

Results:

```
1
2
3
4
While ended when condInt equalled 5
```

Section 8: Built-In Functions

HavaBol contains a variety of built-in functions. These include the following.

Form	**Return Type**
LENGTH(String)	*Int*
SPACES(String)	*Bool*
ELEM(array)	*Int*
MAXELEM(array)	*Int*
print(String, ...)	*N/A*

LENGTH takes one argument between its parentheses and returns the number of characters in the String. As with the other built-in functions that take Strings as arguments, LENGTH coerces the argument, even if it's the result of an expression with a different data type, into a String. See example 4 for its usage.

SPACES returns true if its argument is an empty String - a String like this "" - or a String containing only whitespace; whitespace includes spaces, tabs, and newlines; tab and newline characters may be indicated by "\t" and "\n" respectively within a String. Otherwise, SPACES returns false. See the following example for its usage..

Example:
```
String len = "length";      // here is 6 character String
String wideopen = " \t\n"; // here is a String containing only
                           // whitespace
print("The length of \"length\" is", LENGTH(len));
if SPACES(len) and SPACES(wideopen):
     print("len and empty only contain whitespace");
endif;
if SPACES(len):
     print("len only contains whitespace");
endif;
if SPACES(wideopen):
     print("wideopen only contains whitespace");
endif;
```

Result:
```
The length of "length" is 6
wideopen only contains whitespace
```

ELEM returns the subscript of the highest populated element in an array plus one. This is particularly useful for iterating over an array in a for loop, as shown in example the following example.

MAXELEM returns the number of elements that are declared in an array. This may be less than the size of the array in cases where not every element has been initialized.

Example:
```
Int intArray[5] = 1, , 3, , 5; // only 3 elements declared
Bool boolArray[5] = T, F, T, F, T;
print("The declared length of intArray is", MAXELEM(intArray));

// Iterate from 0 to the last subscript in the array incrementing
// by one.
for a = 0 to ELEM(intArray) by 1:
     if boolArray[a] == T:
          print("intArray[", a, "] =", intArray[a]);
     else:
          print("intArray[", a, "] = UNDECLARED");
     endif;
endfor;
```

Result:
```
The declared length of intArray is 5
intArray[ 0 ] = 1
intArray[ 1 ] = UNDECLARED
intArray[ 2 ] = 3
intArray[ 3 ] = UNDECLARED
intArray[ 4 ] = 5
```

print is unique amongst the built-in functions in that it can take a variable number of arguments and does not return a value. Instead, print serves to output the results of each parameter passed to it, adding a space between parameters and returning to a new line at the end. See the following example for its usage.

Example:
```
String h = "he";
print("1 argument");
print(2, "arguments");
print(1+2, "arguments", h # "re");
```

Result:
```
1 argument
2 arguments
3 arguments here
```

Decisions:

It was decided that rather than printing an array being considered an error, printing an array should print information about it including its size, last declared element index plus one, whether it was bounded or unbounded (when that feature was in production), and its contents.

APPENDIX I: Error Free Test Code

The following code is located in the file ‘p5GoodInput.txt’ which accompanies the interpreter submission and tests the following features:

- variable declarations, assignments, and initializations
- array assignments, declarations, and initializations
- array and String subscripting
- assignment operations
- simple and complex expressions
- order of operations
- if and if/else blocks
- each of the three for-loop types described above
- while loops
- built-in HavaBol functions described above

CODE:

```
// Testing assignments.
print("Assignment tests.");
String helloStr = "Hello World!";
String helloArr[LENGTH(helloStr)];
Int arrTestCounter = 0;
Bool reverseIt = T;

print("helloStr is set to:", helloStr);
print("arrTestCounter is set to:", arrTestCounter);
print("reverseIt is set to:", reverseIt);

// Testing array assignments.
print("\nTesting array assignment to helloArr");
print("Copying chars from helloStr to helloArr.");
for char in helloStr:
     helloArr[arrTestCounter] = char;
     arrTestCounter += 1;
endfor;

print("Printing elements copied to helloArr.");
for elem in helloArr:
     print(elem);
endfor;

// Simple if/else test.
Bool foo = F;
print("Testing simple if else.");
print("foo is set to:", foo);
if foo:
     print("\tCondition is true with foo.");
else:
     print("\tCondition is false with foo.");
endif;

// Test case for both if and for statements.
String sleep = 'Snooz';
Int iCM[6] = 10, 20, 30, 40, 50, 60;
Int n = 2 ;
Int inc = 1;
for i = 0 to n:
    if i/1 == 1:
        print ("outer loop");
    endif;
    for j = 0 to n by inc:
        if j/1 == 1 :
            print ("first inner loop");
        endif;

        for t = 0 to n by inc:
            if t/1 == 1 :
                print("second inner loop");

            endif;
        endfor;
    endfor;
endfor;

// Changing the value of n does not affect the number of
// iterations.
for i = 0 to n:
    print("n = ", n);
    n = n+2;
endfor;

for iVal=0 to 6 by 2:
    print("iVal = \t", iCM[iVal]);
endfor;

// default incrementer is by 1
for iVal=0 to 6:
    print("iVal = \t", iCM[iVal]);
endfor;

// for each element in a array
```

```
for iVal in iCM:
    print("iVal = \t", iVal);
endfor;
// for each char in a string
for ch in sleep :
    print ('ch=', ch);
endfor;

// changing the values of icm
for iVal=0 to 6:
    iCM[iVal] = iCM[iVal] + 5;
    print("iVal = \t", iCM[iVal]);
endfor;

// for each char in a string
for ch in sleep :
    ch = 'a';
    print ('ch=', ch);
endfor;

// Print test cases.
print("have a\t ball\n");
print(1+1, 2, 3);
print(T and F);

// Coercion test cases.
i = 4;
j = 5;
print(i + j);
String str;
str = "1";
print(str + i);
j = str + i;
print(j * str);

// Complex expressions test cases.
String hi = "hi there";
i = 4 + (5 - 6) * 2;
print("i = " # "4 + (5 " # "- 6) " # "* 2");
print("i+2 i+i: ", i+2, i+i);
print("i+2 == 2:", i+2 == 2);
print("i+2 >= 2:", (i+2) >= 2);
print("2 <= i+2:", 2 <= i+2);
i = i ^ i;
print("\"\" # i^4.5  == 256", "" # i^4.5 == 256);
print(hi);
i = LENGTH(hi);
print("LENGTH(\"hi there\") =", i, "or", LENGTH("hi there"));
print(LENGTH(hi) == LENGTH(hi) and T);
print(hi, "only contains spaces or is empty:", SPACES(hi));
print("    ", "only contains spaces or is empty:", SPACES(" "));
print("", "only contains spaces or is empty:", SPACES("") and T);
Int array[] = 1, 2, 3;
print("array[0] + array[1] + array[2] =", array[0] + array[1] +
array[2]);
print("hi[1]", hi[1]);
print(hi[5]#hi[6]#hi[7], hi[3]#hi[4]#hi[5],
hi[6]#hi[1]#hi[3]#hi[5], hi[1], hi[0]#hi[1]#hi[5]);
String hilo = hi[-1]#hi[-2]#hi[-3]#hi[-4]#hi[-5];
print(hilo);
print("ELEM(array):", ELEM(array), "MAXELEM(array):",
MAXELEM(array));
Int other[] = 4, 5, 6;
print(array);
print(other);
i = 1 + 1;
array[i] = i;
print(array);

// While loop and String subscript test.
Int whileTestCounter = LENGTH(helloStr);
String reverseHelloStr = "";
print("\nWhile loop to copy reversed helloStr with String
subscripting.");
reverseIt = whileTestCounter >= 0;
while reverseIt:
      reverseHelloStr = reverseHelloStr #
      helloStr[whileTestCounter - 1];
      whileTestCounter -= 1;
      reverseIt = whileTestCounter >= 0;
endwhile;
print("\"" # helloStr # "\" reversed is \"" # reverseHelloStr #
"\".");

String forStr = "for";
String reignStr = "reign";
String foreignStr = forStr;
foreignStr[2] = reignStr;
print("\nTesting String subscripting in assignment.");
print("forStr is set to:", forStr);
print("reignStr is set to:", reignStr);
print("forStr[2] = reignStr yields:", foreignStr);
```

APPENDIX II: Error Case Tests

Section AII.1: Assignment Errors

1. Declaring variable with wrong type.
   CODE:
   ```
   Int t = F;
   ```
   RESULT:
   ```
   Line 2 havabol.ResultValueConversionException: cannot
   convert Bool to integer type, File:
   docs/tests/testingoutput.txt
        at havabol.Parse.error(Parse.java:1298)
        at havabol.Parse.assign(Parse.java:721)
        at havabol.Parse.assignmentStmt(Parse.java:536)
        at havabol.Parse.parseStmt(Parse.java:141)
        at havabol.HavaBol.main(HavaBol.java:50)
   ```

2. Accessing uninitialized variables.
   CODE:
   ```
   Int k;
   print(k);
   ```
   RESULT:
   ```
   Line 1 attempting to access uninitialized variable, File:
   docs/testcases/ErrorTestcases.txt
        at havabol.Parse.error(Parse.java:1288)
        at havabol.Parse.getValueOfToken(Parse.java:620)
        at havabol.Parse.expr(Parse.java:789)
        at havabol.Parse.print(Parse.java:1467)
        at havabol.Parse.callBuiltInFunction(Parse.java:1428)
        at havabol.Parse.parseStmt(Parse.java:117)
        at havabol.HavaBol.main(HavaBol.java:50)
   ```

3. Attempting to mask an previously declared variable.
   CODE:
   ```
   Int t = 3;
   print(t);
   Int t = 4;
   print(t);
   ```
   RESULT:
   ```
   3
   Line 3 variable "t" was already declared, File:
   docs/tests/testingoutput.txt
        at havabol.Parse.error(Parse.java:1297)
        at havabol.Parse.declareStmt(Parse.java:1098)
        at havabol.Parse.parseStmt(Parse.java:56)
   ```

```
                at havabol.HavaBol.main(HavaBol.java:50)
```

Section AII.2: Array and Subscripting Errors

1. Assigning more values to an array than its declared size.
   CODE:
   ```
   String strArr[3] = "cat", "dog", "snake", "mouse";
   ```
   RESULT:
   ```
   Line 2 assigned more values to array than its declared size,
   File: docs/tests/testingoutput.txt
        at havabol.Parse.error(Parse.java:1297)
        at havabol.Parse.arrayAssignmentStmt(Parse.java:331)
        at havabol.Parse.parseStmt(Parse.java:137)
        at havabol.HavaBol.main(HavaBol.java:50)
   ```

2. Assigning a value to an array that cannot be converted to its declared type.
   CODE:
   ```
   Int intArr[] = 2, 3, T;
   ```
   RESULT:
   ```
   havabol.ResultValueConversionException: cannot convert Bool
   to integer type
        at havabol.ResultValue.convertType(ResultValue
        .java:425)
        at havabol.Parse.arrayAssignmentStmt(Parse.java:311)
        at havabol.Parse.parseStmt(Parse.java:137)
        at havabol.HavaBol.main(HavaBol.java:50)
   ```

3. Out of bounds in a fixed size array.
   CODE:
   ```
   Int intArr[] = 2, 3, "3.0";
   print(intArr[3]);
   ```
   RESULT:
   ```
   Line 2 Out of bounds error with index of 3, File:
   docs/tests/testingoutput.txt
        at havabol.Parse.error(Parse.java:1303)
        at havabol.Parse.expr(Parse.java:812)
        at havabol.Parse.print(Parse.java:1482)
        at havabol.Parse.callBuiltInFunction(Parse.java:1443)
        at havabol.Parse.parseStmt(Parse.java:117)
        at havabol.HavaBol.main(HavaBol.java:50)
   ```

4. String index out of bounds.
   CODE:
   ```
   String helloStr = "Hello World!";
   String helloChar = helloStr[12];
   ```
   RESULT:
   ```
   Line 2 String index out of bounds error with index of 12,
   File: docs/tests/testingoutput.txt
        at havabol.Parse.error(Parse.java:1309)
        at havabol.Parse.expr(Parse.java:855)
        at havabol.Parse.assignmentStmt(Parse.java:527)
        at havabol.Parse.parseStmt(Parse.java:141)
        at havabol.HavaBol.main(HavaBol.java:50)
   ```

Section AII.3: Expression and Operation Errors

1. Invalid expression: ‘and’ lacks lefthand operand.
   CODE:
   ```
   print(and F);
   ```
   RESULT:
   ```
   Line 1 Invalid expression, File:
   docs/testcases/ErrorTestcases.txt
        at havabol.Parse.error(Parse.java:1288)
        at havabol.Parse.expr(Parse.java:972)
        at havabol.Parse.print(Parse.java:1467)
        at havabol.Parse.callBuiltInFunction(Parse.java:1428)
        at havabol.Parse.parseStmt(Parse.java:117)
        at havabol.HavaBol.main(HavaBol.java:50)
   ```

2. Operator ‘+’ lacks righthand operand.
   CODE:
   ```
   Int i = 1 +;
   ```
   RESULT:
   ```
   Line 1 Invalid expression, File:
   docs/testcases/ErrorTestcases.txt
        at havabol.Parse.error(Parse.java:1288)
        at havabol.Parse.expr(Parse.java:972)
        at havabol.Parse.assignmentStmt(Parse.java:526)
        at havabol.Parse.parseStmt(Parse.java:141)
        at havabol.HavaBol.main(HavaBol.java:50)
   ```

3. ‘or’ righthand operand is not correct type.
   CODE:
   ```
   print(T and F or 5);
   ```
   RESULT:

```
Line 1 havabol.ImproperOperationException: Operand's
datatype is inappropriate for or operations, File:
docs/testcases/ErrorTestcases.txt
     at havabol.Parse.error(Parse.java:1293)
     at havabol.Parse.expr(Parse.java:998)
     at havabol.Parse.print(Parse.java:1472)
     at havabol.Parse.callBuiltInFunction(Parse.java:1433)
     at havabol.Parse.parseStmt(Parse.java:117)
     at havabol.HavaBol.main(HavaBol.java:50)
```

4. Invalid operation on an array.
   CODE:
   ```
   Int iCM[5] = 1, 2, 3, 4, 5;
   Int k = 0;
   k = iCM + k;
   ```
   RESULT:
   ```
   Line 3 Cannot perform + operation on an array, File:
   docs/testcases/ErrorTestcases.txt
        at havabol.Parse.error(Parse.java:1293)
        at havabol.Parse.expr(Parse.java:809)
        at havabol.Parse.assignmentStmt(Parse.java:526)
        at havabol.Parse.parseStmt(Parse.java:141)
        at havabol.HavaBol.main(HavaBol.java:50)
   ```

5. Invalid token found in expression.
   CODE:
   ```
   Int i = 1 + 3 $ 2;
   ```
   RESULT:
   ```
   havabol.ScannerTokenFormatException: SCANNER ERROR: token on
   line 15 at column 14 cannot be classified
        at havabol.Scanner.createNextToken(Scanner.java:159)
        at havabol.Scanner.getNext(Scanner.java:82)
        at havabol.Parse.expr(Parse.java:931)
        at havabol.Parse.assignmentStmt(Parse.java:526)
        at havabol.Parse.parseStmt(Parse.java:141)
        at havabol.HavaBol.main(HavaBol.java:50)
   ```

6. Lefthand operand of ‘and’ is not a Bool.
   CODE:
   ```
   if (1 + 2 and 3 < 2):
        print("This will throw an error");
   endif;
   ```
   RESULT:
   ```
   Line 1 Line 1 Invalid expression. Expected operands or
   operators are missing in expression., File:
   ```

```
docs/testcases/ErrorTestcases.txt, File:
docs/testcases/ErrorTestcases.txt
     at havabol.Parse.error(Parse.java:1297)
     at havabol.Parse.expr(Parse.java:1002)
     at havabol.Parse.ifStmt(Parse.java:1185)
     at havabol.Parse.parseStmt(Parse.java:74)
     at havabol.HavaBol.main(HavaBol.java:50)
```

Section AII.4: If Statement Errors

1. Missing ‘if’ token on line 2.
   CODE:

```
Bool foo = F;
foo:
     print("hi");
else:
     print("bye");
endif;
```

   RESULT:

```
Line 3 assignment expects operator, File:
docs/tests/testingoutput.txt
     at havabol.Parse.error(Parse.java:1298)
     at havabol.Parse.assignmentStmt(Parse.java:510)
     at havabol.Parse.parseStmt(Parse.java:141)
     at havabol.HavaBol.main(HavaBol.java:50)
```

2. Missing ‘:’ at end of line 2.
   CODE:

```
Bool foo = F;
if foo
     print("hi");
else:
     print("bye");
endif;
```

   RESULT:

```
Line 3 Cannot perform operation with print function, File:
docs/tests/testingoutput.txt
     at havabol.Parse.error(Parse.java:1298)
     at havabol.Parse.expr(Parse.java:916)
     at havabol.Parse.ifStmt(Parse.java:1186)
     at havabol.Parse.parseStmt(Parse.java:74)
     at havabol.HavaBol.main(HavaBol.java:50)
```

3. Missing ‘:’ after ‘else’ token on line 4.

CODE:
```
Bool foo = F;
if foo:
     print("hi");
else
     print("bye");
endif;
```
RESULT:
```
     Line 5 expected ':' after 'else', File:
     docs/tests/testingoutput.txt
          at havabol.Parse.error(Parse.java:1298)
          at havabol.Parse.ifStmt(Parse.java:1226)
          at havabol.Parse.parseStmt(Parse.java:74)
          at havabol.HavaBol.main(HavaBol.java:50)
```

4. Missing ‘endif’ token.

CODE:
```
Bool foo = F;
if foo:
     print("hi");
else:
     print("bye");
```
RESULT:
```
bye
Line 6 expected endif, File: docs/tests/testingoutput.txt
     at havabol.Parse.error(Parse.java:1298)
     at havabol.Parse.ifStmt(Parse.java:1234)
     at havabol.Parse.parseStmt(Parse.java:74)
     at havabol.HavaBol.main(HavaBol.java:50)
```

5. Missing ‘;’ after ‘endif’ token.

CODE:
```
Bool foo = F;
if foo:
     print("hi");
else:
     print("bye");
endif
```
RESULT:
```
bye
Line 8 expected ';' after endif, File:
docs/tests/testingoutput.txt
     at havabol.Parse.error(Parse.java:1298)
     at havabol.Parse.ifStmt(Parse.java:1243)
     at havabol.Parse.parseStmt(Parse.java:74)
```

```
                    at havabol.HavaBol.main(HavaBol.java:50)
```

Section AII.5: For Statement Errors

1. The ‘to’ token is missing on line 3.
   CODE:
```
        Int iCM[6] = 10, 20, 30, 40, 50, 60;
        Int n = 6;
        for iVal = 0 by 1:
             print("iVal = \t", iCM[iVal]);
        endfor;
```
   RESULT:
```
        Line 3 format : for cv = sv to limit by incr:
        expected 'to' after after cv = sv, File:
        docs/tests/testingoutput.txt
             at havabol.Parse.error(Parse.java:1371)
             at havabol.Parse.forStmt(Parse.java:2017)
             at havabol.Parse.parseStmt(Parse.java:66)
             at havabol.HavaBol.main(HavaBol.java:50)
```

2. The ‘in’ token is missing on line 2.
   CODE:
```
        Int n = 6;
        for iVal  sleep:
             print("iVal = \t", iVal);
        endfor;
```
   RESULT:
```
        Line 2 format : for cv = sv to limit by incr:
          for cv in <string/array> by incr:
          for cv in <string/array>, File:
          docs/tests/testingoutput.txt
             at havabol.Parse.error(Parse.java:1371)
             at havabol.Parse.forStmt(Parse.java:2283)
             at havabol.Parse.parseStmt(Parse.java:66)
             at havabol.HavaBol.main(HavaBol.java:50)
```

3. Missing control variable on line 3.
   CODE:
```
        String sleep = "snooz"
        Int n = 6;
        for in sleep:
             print("iVal = \t", iVal);
        endfor;
```
   RESULT:

```
Line 3 format : for cv = sv to limit by incr:
  for cv in <string/array> by incr:
  for cv in <string/array>, File:
  docs/tests/testingoutput.txt
      at havabol.Parse.error(Parse.java:1371)
      at havabol.Parse.forStmt(Parse.java:2283)
      at havabol.Parse.parseStmt(Parse.java:66)
      at havabol.HavaBol.main(HavaBol.java:50)
```

4. Missing ‘endfor’ token at end of loop.
CODE:
```
Int iCM[6] = 10, 20, 30, 40, 50, 60;
for iVal in iCM:
     print("iVal = \t", iVal);
```
RESULT:
```
iVal =      10
Line 4 invalid syntax at the end of a for loop: expected
'endfor' at the end of a for loop, File:
docs/tests/testingoutput.txt
     at havabol.Parse.error(Parse.java:1371)
     at havabol.Parse.forStmt(Parse.java:2195)
     at havabol.Parse.parseStmt(Parse.java:66)
     at havabol.HavaBol.main(HavaBol.java:50)
```

5. Missing ‘;’ after ‘endfor’ token at end of for loop.
CODE:
```
Int iCM[6] = 10, 20, 30, 40, 50, 60;
for iVal in iCM:
     print("iVal = \t", iVal);
endfor
```
RESULT:
```
iVal =      10
iVal =      20
iVal =      30
iVal =      40
iVal =      50
iVal =      60
Line 5 invalid syntax at end of for loop: expected ';' but
received '', File: docs/tests/testingoutput.txt
     at havabol.Parse.error(Parse.java:1371)
     at havabol.Parse.forStmt(Parse.java:2297)
     at havabol.Parse.parseStmt(Parse.java:66)
     at havabol.HavaBol.main(HavaBol.java:50)
```

6. Missing ‘=’ on line 2 on assignment of value to *cv*.

CODE:
```
Int iCM[6] = 10, 20, 30, 40, 50, 60;
for iVal 0 to 6 by 2:
     print("iVal = \t", iCM[iVal]);
endfor;
```
RESULT:
```
Line 2 format : for cv = sv to limit by incr:
  for cv in <string/array> by incr:
  for cv in <string/array>, File:
  docs/tests/testingoutput.txt
      at havabol.Parse.error(Parse.java:1371)
      at havabol.Parse.forStmt(Parse.java:2283)
      at havabol.Parse.parseStmt(Parse.java:66)
      at havabol.HavaBol.main(HavaBol.java:50)
```

Section AII.6: While Loop Errors

1. While condition cannot be evaluated to boolean

CODE:
```
Int cond = 4;
while(cond):
     print("Hello World!");
endwhile;
```
RESULT:
```
Line 3 while condition could not be evaluated to boolean,
File: docs/tests/testingoutput.txt
     at havabol.Parse.error(Parse.java:1309)
     at havabol.Parse.whileStmt(Parse.java:1840)
     at havabol.Parse.parseStmt(Parse.java:61)
     at havabol.HavaBol.main(HavaBol.java:50)
```

2. Missing ‘;’ after ‘endwhile’ token.

CODE:
```
Int cond = 2;
while(cond > 0):
     print("Hello World!");
     cond -= 1;
endwhile
```
RESULT:
```
Hello World!
Hello World!
Line 6 invalid syntax at end of while loop: expected ';' but
received '', File: docs/tests/testingoutput.txt
     at havabol.Parse.error(Parse.java:1309)
```

```
        at havabol.Parse.whileStmt(Parse.java:1893)
        at havabol.Parse.parseStmt(Parse.java:61)
        at havabol.HavaBol.main(HavaBol.java:50)
```

Section AII.7: Built-In Function Errors

1. Argument to ELEM() function is not an array.
   CODE:
   ```
   Int i = 1;
   ELEM(i);
   ```
   RESULT:
   ```
   Line 2 ELEM function takes an array as an argument, File:
   docs/testcases/ErrorTestcases.txt
           at havabol.Parse.error(Parse.java:1297)
           at havabol.Parse.elem(Parse.java:1686)
           at havabol.Parse.callBuiltInFunction(Parse.java:1451)
           at havabol.Parse.parseStmt(Parse.java:117)
           at havabol.HavaBol.main(HavaBol.java:50)
   ```

2. Argument to LENGTH() is not initialized.
   CODE:
   ```
   Int i;
   print(LENGTH(i));
   ```
   RESULT:
   ```
   Line 2 attempting to access uninitialized variable, File:
   docs/testcases/ErrorTestcases.txt
           at havabol.Parse.error(Parse.java:1297)
           at havabol.Parse.getValueOfToken(Parse.java:620)
           at havabol.Parse.expr(Parse.java:793)
           at havabol.Parse.length(Parse.java:1561)
           at havabol.Parse.callBuiltInFunction(Parse.java:1442)
           at havabol.Parse.expr(Parse.java:918)
           at havabol.Parse.print(Parse.java:1476)
           at havabol.Parse.callBuiltInFunction(Parse.java:1437)
           at havabol.Parse.parseStmt(Parse.java:117)
           at havabol.HavaBol.main(HavaBol.java:50)
   ```