**Table of Contents**

Introduction

Getting started

Chapter 1 Data types

1.1 Int

1.2 Float

1.3 String

1.4 Bool

Chapter 2 Arrays (including unbounded)

2.1 Declarationand Unbounded arrays and more

2.2 Iteration

Chapter 3 Operations (including infix vs prefix, IN, NOTIN, additional numeric assignment)

3.1 Expressions

3.2 Operands

3.3 Operators

3.4 Operator precedence

3.5 Assignment operator

Chapter 4 : Slices

Chapter 5 Flow Control (if, while, for, select, break, continue, for tokenizer)

5.1 if

5.2 while

5.3 for loops

5.4 counting for

5.5 foreach

5.6 tokenizer loop

5.7 Select/When

5.8 Break/Continue

Chapter 6 Builtin Functions

6.1 Print

6.2 Length

6.3 Spaces

6.4 Elem

6.5 MaxElem

Chapter 7 Programmer-Defined Functions

Chapter 8 Test Cases

8.1 Counting For Loop Valid Test Cases

8.2 Counting For Loop Error Test Cases

8.3 Nested Counting For Loop Error Test Cases

8.4 Foreach Loop Error Test Cases

8.5 Nested Foreach Loop Test Cases

8.6 For Tokenizer Loop Test Cases

8.7 Nested For Tokenizer Loop Test Cases

8.8 Assignment Valid Test Cases

8.9 Array Assignment Error Test Cases

8.10 Assignment Error Test Cases

8.11 Built-In Functions Valid Test Cases

8.12 Built-In Functions Error Test Cases

8.13 Expression Valid Test Cases

8.14 Expression Error Test Cases

8.15 If Statement Valid Test Cases

8.16 If Statement Error Test Cases

8.17 Nested If In Else Error Test Cases

8.18 While Statement Valid Test Cases

8.19 While Statement Error Test Cases

8.20 Nested While Statement Error Test Cases

**Introduction**

Let’s start with a brief introduction to HavaBol. It is a strongly typed interpreted language. The interpreter parses and compiles all HavaBol source code into Java bytecode. The scope of this language is global. (more details in each section) This is a simple language with basic control flow mechanisms : for, foreach, if, while and just a handful of builtin functions. The syntax is very Python like but what this language can do as far as any complicated concepts is very limited. This would be a good beginner language. The aim of this document is to serve as a reference to the programmer on how to use HavaBol. Hope you enjoy and make sure you have a ball .

**Getting Started**

First Program:

As with any programing language the first program will be ‘hello world’. Heres that program in Havabol:

print(“Hello World!\n”); // I’m a comment and the interpreter ignores me!

HavaBol supports one line comments. They are started will // as shown above.

‘print’ is a builtin function that takes in string literals and other intialized variables as arguments. Its prints to standard out (aka the screen by default)

A string literal is indicated by the double quotes: “String literal” is a string literals

The backslash n is a special character which will be discussed in chapter one when going over strings. (it just means add a newline)

The print builtin function by default will ALWAYS add a newline, even if not specified.

The above program prints out: ‘Hello World!’ followed by 2 newline characters.

A quick comment about scope. HavaBol used only global scope for all variables so if you have a variable already initialized to a value, you can redeclare it so you don’t have to keep renaming variables. Example below:

Int a = 2 \* 10;

print(a); // prints ‘20’

…

…

Int a; // redeclared ‘a’ . its initial value is null.

print(a); // prints null

**Chapter One : Data Types**

**Section 1.1 type Int**

This section describes the type ‘Int’. ‘Int’ represents the type integer. ‘Int’ is just the keyword used upon declaration of a variable to indicate that the type is an integer. An integer,in this language is 4 bytes. An integer’s value can range is from -2,147,483,648 to 2,147,483,647.

Example declaration of a variable called num of type Int being initialized to value 100:

Int num = 100;

Example of declaration without initialization:

Int num;

Format for declaration: type varName; or type varName ‘=’ expression;

\*\*Note\*\* A concise definition of expressions will be outlined in Chapter 3.

The type is required and obviously the variable name is required and a semicolon is required as well.

Initialization does not have to be done upon declaration. Regardless of where a variable is declared in HavaBol, its scope is global. This language has no other scope.

\*\*Note: ‘Int’ can be used to declare a simple variable or to declare an array of type int.

Example: Int inums[3] = 2, 4, 8;

\*\* Arrays will be covered in detail in Chapter 2.

\*\*NOTE: For ALL types, before any operations can be performed a variable must be initialized first. More on this in Chapter 3.

A note about type casting for type Int.

The right hand side will always be cast to the type of the left handside (of an equals for example) when possible.

i.e. Int num = 3.14; will result in the value 3 being assigned to num as it was cast to an integer before assignement.

rules for Int casting:

Int → Float YES , will cast float to int, rounding the value down to the nearest integer

Int → String YES /NO, will cast the string to a an Int as long as the String is takes the value of an Int. ie. “10” can be cast to an Int. “Chris” cannot be cast to an Int.

\*\*Note: the String “3.14” can NOT be cast to Integer because it is not an string literal which is an integer.

Int → Bool NO, will cause an error

**Section 1.2 type Float**

This section describes the type ‘Float’. ‘Float’ represents a double-precision 64 bit IEEE 754 floating point. ‘Float’ is just the keyword used upon declaration of a variable to indicate that the type is a double value. Type ‘Float’ in this language is 8 bytes.

Example declaration of a variable called fnum of type Float being initialized to value 3.14:

Float fnum = 3.14;

Example of declaration without initialization:

Float fnum;

Format for declaration: type varName; or type varName ‘=’ expression;

\*\*Note\*\* A concise definition of expressions will be outlined in Chapter 3.

The type is required and obviously the variable name is required and a semicolon is required as well.

Initialization does not have to be done upon declaration. Regardless of where a variable is declared in HavaBol, its scope is global.

\*\*Note: ‘Float’ can be used to declare a simple variable or to declare an array of type Float.

Example: Float nums[3] = 3.14, 2.22, 1.1;

\*\* Arrays will be covered in detail in Chapter 2.

\*\*NOTE: For ALL types, before any operations can be performed a variable must be initialized first. More on this in Chapter 3.

A note about type casting for type Float.

The right hand side will always be cast to the type of the left handside (of an equals for example) when possible.

i.e. Float myFloat = 3; will result in the value 3.00 being assigned to num as it was cast to a Float before assignment. (2 decimal points added)

rules for Float casting:

Float → Int YES , will cast int to a float, adding a decimal point and 2 zeros

Float → String NO, will cause an error, even if the string is an Float. I.e Float num = “10.1”;

Float → Bool NO, will cause an error

**Section 1.3 type String**

This section describes the type ‘String’. It is not abbreviated at all. The string is actually the Java String class behind the scenes. ‘String’ is just the keyword used upon declaration of a variable to indicate that the type is a String . The size of type string is not set in stone. As it is Java string class its size starts of at 36 bytes (due to necessary references and other values) plus 2 bytes \* length of the string. A String can be almost any ASCII character plus a few non-printable ones. The non-printables include:

\n - newline character

\t - the tab character

\a - the alarm bell character

Example declaration of a variable called str of type String being initialized:

String str = “Hello\tWorld!\n”;

Example of declaration without initialization:

String str;

Format for declaration: type varName; or type varName ‘=’ string literal ;

The type is required and obviously the variable name is required and a semicolon is required as well.

Initialization does not have to be done upon declaration. Regardless of where a variable is declared in HavaBol, its scope is global. This language has no other scope.

Type String also can be indexed like an array. For example:

String name = “Chris\n”;

name[5] = “t”;

print(name); // this will print ‘Christ’

Also String is the only scalar type that is iterable. That means you can use a for counter loop or a foreach loop to iterate thru the individual characters in the String type.

\*\*\*Note: more on control flow (for loops and such) in chapter 4.

\*\*Note: ‘String’ can be used to declare a simple variable or to declare an array of type String.

Example: String names[3] = “Chris”, “Matt”, “Miguel”;

\*\* Arrays will be covered in detail in Chapter 2.

\*\*NOTE: For ALL types, before any operations can be performed a variable must be initialized first. More on this in Chapter 3.

A note about type casting for type String.

The right hand side will always be cast to the type of the left handside (of an equals for example) when possible.

i.e. String str = 10; will result in the value “10” (a string) being assigned to str as it was cast to a String before assignment.

rules for String casting:

String → Int YES , will cast Int to a String

String → Float YES, will cast Float to a String

String → Bool YES cast Bool to String

String also support concatenation using the ‘#’ operator.

Example:

String first = “Chris”;

String last = “Buckner”;

String name = first # ” “ # last;

print(name);

Output:

Chris Buckner

Explanation:

Both variables first and last were declared. Then upon declaration of name, ‘first’ was concatenated with string literal “ “ (one empty space) and that was concatenated with ‘last’.

Finally ‘name’ was printed out.

**Section 1.4 type Bool**

This section describes the type ‘Bool’. ‘Bool’ represents the type boolean. Aka true or false. ‘Bool’ is just the keyword used upon declaration of a variable to indicate that the type is a boolean. ‘Bool’ s values are either true or false. True is represented is HavaBol by ‘T’ and false is represented by ‘F’ (no quotes needed at all).

Example declaration of a variable called ‘b’ of type Bool being initialized to value true:

Bool b = T;

Other Bool examples:

Bool b2 = 10 > 2; //Bool b2 = ‘expression that can be evaluated to True or False’

\*Note: b2 was assigned the value T because the expression evaluated to true.

Example of declaration without initialization:

Bool b;

Format for declaration: type varName; or type varName ‘=’ expression;

\*\*Note\*\* A concise definition of expressions will be outlined in Chapter 3.

The type is required and obviously the variable name is required and a semicolon is required as well.

Initialization does not have to be done upon declaration. Regardless of where a variable is declared in HavaBol, its scope is global. This language has no other scope.

\*\*Note: ‘Bool’ can be used to declare a simple variable or to declare an array of type Bool.

Example: Bool boolVals[3] = T, 10 > 2, F;

\*\* Arrays will be covered in detail in Chapter 2.

\*\*NOTE: For ALL types, before any operations can be performed a variable must be initialized first. More on this in Chapter 3.

rules for Bool casting:

Bool → Int NO, will error, not possible

Bool → Float NO, will error, not possible

Bool → String YES/NO possible if String value is either “T” or “F” ; will ALL other Strings it will error

**Section 1.5 type Date**

**\*\*\*Date data type is not supported in HavaBol**

**Chapter 2 Arrays**

**Section 2.1 Declaration, Unbounded arrays and more**

This section describes the declaration for all types of arrays. Arrays can be of any type: Int, Float, String, Bool. Unlike a simple int, bool, string or float, an array is not a scalar variable. It is the only builtin data structure that HavaBol has.

\*\* A Note on arrays before diving in:

- The same typecasting rules apply to arrays as with regular scalars. Refer to previous chapter

- Arrays can be declared of a fixed size or can be unbounded (dynamic).

- Array to array assignment of all elements in array is also supported

- An array is iterable and can be iterated over using a for loop.

- ‘unbound’ is the keyword to declare an unbounded array

- a fixed size array has out of bounds checking and will throw an error if the user tries to index something that larger than the size of the array

Array declaration examples:

Int array[10]; - declared to be a fixed size array, type Int of 10 elements, no elements initialized

Float array[unbound]; - declared to be an unbounded array of type Float, array has one element initialized to null

Int nums[] = 1, 2, 3, 4; - declared to be a fixed size array of 4 elements each of which initialized

Int nums[4] = 1, 2, 3, 4; – the same as above but being more explicit about size. If you dont specify a size in the brackets upon declaration HavaBol will count the number of elements being initialized and set the size to that.

Int nums[10] = 1, 3.14, 2.22; - declared to be a fixed array of size 10. the first 3 elements are initialized but both floats (3.14 and 2.22) are cast to Ints before initialization. (elements are 1,3,2)

String elems[10] = “cat”, 10, 3.14; - String type array, fixed size of size 10 declared and three elements initialized: a string literal (“cat”), an integer and a float. All values were cast to type String.

String elems[10] = 1, “hello”, T, “bye”;

\*Note\* You cannot declare a fixed array of size 0 as it will also throw an error.

Example:

Int array[0]; // this will error out!

Arrays also support array to array copying in one line.

\*\*Note: When copying array to array. The types can be different but must conform to the typecasting conventions stated in chapter 1.

Examples:

Int aM[unbound] = 10, 100, 1000;

Float aF[unbound] = 3.14, 2.22, 5.67;

aM = aF;

print(aM);

Output:

[3, 2, 5]

Explanation:

Each individual value was casted to an Int and then each element in aM was set to the corresponding casted values.

Int nums[10] = 1,2,3,4,5; (size 10)

Int moreNums[] = 6,7,8,9,10,11,12,13,14,15; (also size 10)

nums = moreNums;

Since nums is of size 10 , all 10 values of moreNums will be copied over to nums so if:

print(nums);

was executed.

Output

[6, 7, 8, 9, 10, 11, 12, 13, 14 ,15]

If arrays of different sizes:

Example:

Int array[3] = 1,2,3;

Int array2[5] = 5,4,3,2,1;

array = array2;

Only the first 3 values of array2 would be copied to array. If we were to print(array);

the output would be:

[5, 4, 3]

\*\* Note: if you try to assign an array to a scalar, this will cause an error as this is NOT possible to do, although if you do the opposite this is allowed and will just default all the elements of the array to the scalar that you want.

Examples:

Int nums[10] = 1,2,3,4,5,6;

Int var = nums; // this will error out because ‘var’ is a scalar variable and NOT an array

Int mydefault = 1024;

Int nums[] = 1,2,3,4,5,6;

nums = mydefault; // this is valid, all elements of nums are now 1024

How to set a particular value of an array? Indexing.

Heres an example:

Int arr[] = 1,2,3,4,5; // size is automatically set to size 5

arr[0] = 99999; // set first element to all 9’s using the first index

arr[-1] = 99999; // set last element to all 9’s using the last index

print(arr);

Output:

[99999, 2, 3, 4, 99999]

Array indexing starts at zero , so index zero is the first element and we can access the last element we can access is by using index ‘-1’.

Last note on indexing: HavaBol will not let you index out of bounds on a fixed size array, although there is no ‘out of bounds’ on an unbounded array so it will just ‘grow’ the unbounded array. (refer to unbounded arrays earlier in chapter for more detail)

Example using fixed array and indexing too far:

Int arr[5] = 1, 2, 3, 4, 5;

arr[10] = 11; // this will throw an error because we are trying to set an element out of bounds of this array. Bounds checking implemented.

We can have ‘holes’ in arrays on certain conditions. When I say ‘holes’ I mean that the value in that index will be null.

Examples:

Int array[unbound]; - declare unbounded

array[10] = 1000; - set index 10 to 1000

The first 9 values of this unbounded array would be null’s while only one index would have value 1000(the last index)

\*Note: this is a way to grow an unbounded array.

Int array[10] = 1,2,3; - declared to size 10

array[5] = 1000; - set index 5 to value 1000

If we print out array we get:

[1, 2, 3, null, null, 1000]

We set the first 3 values to numbers and then set the sixth value (index 5) to 1000. There are 2 ‘holes’ (null values) in the printed values.

Arrays in HavaBol can also be default to a particular value.

Examples:

Int array[10];

array = 1000;

Result: All 10 elements of ‘array’ will be set to 1000.

When setting a default value for unbounded arrays that have not been initialized, the first , and only, element of the array will be set to that default value.

Example:

Int arr[unbound];

arr = 100;

print(arr);

Output:

[100]

**Section 2.2 Iteration**

Arrays are indeed iterable and you can iterate thru them most commonly using a for counting loop or a foreach loop.

Example1:

Int nums[10] = 1,2,3,4;

for val in nums:

print(val);

endfor;

Output:

1

2

3

4

Explanation:

In a foreach loop, there is no enumeration and you can only print out the values in the array and it will print out all the values up to the last initialized value.

Example 2:

Int size = 5

Int nums[size] = 2, 4, 8;

for j=0 to size:

print(“j= “, j, “; val = “, nums[j]);

endfor;

Output:

j= 0; val = 2

j= 1; val = 4

j= 2; val = 8

j= 3; val = null

j= 4; val = null

Explanation:

In a counting for loop, there is enumeration (the j variable) and it will try to iterate up to what was specified. Not inclusive. ie. size. it has a value of five but loop only iterated until value 4.

It is possible to go out of bounds using a counting for loop . Also the counting variable can be used in the for loop as above.

**Chapter 3 Operations**

**Section 3.1 Expressions**

In HavaBol expressions include arithmetic, conditional, as well as some unary operations. Expressions are represented in infix notation.

Example of basic expression format:

*Operand Operator Operand*

Example of basic unary expression format:

*Operator Operand*

**Section 3.2 Operands**

Operands in HavaBol can range from a variety of options.

**Section 3.2.1 Primitive Data Types**

1. *Integer*
2. *Float*
3. *String*
4. *Boolean*

(See Chapter 1 for more information)

**Section 3.2.2 Arrays**

1. Array Object
2. Array Index Reference
3. String Index Reference

(See Chapter 2 for more information)

**Section 3.2.3 Identifiers**

1. Variables that can either be single values or array names

**Section 3.2.4 Built-In Functions**

1. LENGTH
2. SPACES
3. ELEM
4. MAXELEM

(See Chapter 6 for more information)

**Section 3.3 Operators**

**Section 3.3.1 Arithmetic Operators**

1. Addition “+”
   1. *Operand* + *Operand*
   2. Adds the right Operand to the left Operand
   3. Ex: 1 + 1 = 2
2. Subtraction “-”
   1. *Operand* - *Operand*
   2. Subtracts the right Operand from the left Operand
   3. Ex: 2 - 1 = 1
3. Multiplication “\*”
   1. *Operand* \* *Operand*
   2. Multiplies the left Operand by the right Operand
   3. Ex: 2 \* 2 = 4
4. Division “/”
   1. *Operand* / *Operand*
   2. Divides the left Operand by the right Operand
   3. Ex: 4 / 2 = 2
5. Exponentiation “^”
   1. *Operand* ^ *Operand*
   2. Raises the left Operand to the power of the right Operand
   3. Ex: 2 ^ 4 = 16
6. Unary Minus (Numeric Negation) “-”
   1. - *Operand*
   2. Negates the right Operand
   3. Ex: - 5 = -5

Arithmetic operators return type is based off the left operand type. With the exception of the Unary Minus since there is only one operand. These operators only accept valid numeric coercions as operands. (see Chapter 1 for coercion rules).

**Section 3.3.2 Conditional Operators**

1. Less Than “<”
   1. Operand < Operand
   2. Returns true if the left Operand is less than the right Operand
   3. Ex: 1 < 100 = T
2. Less Than Equals “<=”
   1. Operand <= Operand
   2. Returns true if the left Operand is less than or equal to the right Operand
   3. Ex: 100 <= 100 = T
3. Greater Than “>”
   1. Operand > Operand
   2. Returns true if the left Operand is greater than the right Operand
   3. Ex: 100 > 1 = T
4. Greater Than Equals “>=”
   1. Operand >= Operand
   2. Returns true if the left Operand is greater than equal the right Operand
   3. Ex: 100 >= 100 = T
5. Equals “==”
   1. Operand == Operand
   2. Returns true if the left Operand is equal to the right Operand
   3. Ex: 1 == 1 = T
6. Not Equals “!=”
   1. Operand != Operand
   2. Returns true if the left Operand is not equal to the right Operand
   3. Ex: 1 != 100 = T

The above conditional operators behave differently based on the left Operand type. If the left Operand is a numeric type then it performs a true numeric comparison between the two values. If the left Operand is a string type then it performs a lexicographical comparison between the two values.

1. And “and”
   1. Operand and Operand
   2. Returns true if the left Operand and the right Operand are true
   3. No Short Circuiting occurs
   4. Ex: T and T = T
2. Or “or”
   1. Operand or Operand
   2. Returns true if the left Operand or the right Operand are true
   3. No Short Circuiting occurs
   4. Ex: T or F = T
3. Unary Not (Boolean Negation) “not”
   1. not Operand
   2. Returns the negation of the right Operand
   3. Ex: not F = T
4. **In and NotIn “in” “notin”**
   1. **These Conditional Operations are not supported in Havabol.**

Conditional Operators return type is boolean. Comparison operations do not accept operands that of type Boolean. Logical operations only accept operands that are type Boolean or Strings that can properly be coerced into a Boolean. (see Boolean Coercion in Chapter 1 for more information).

**Section 3.3.3 Concatenation Operator “#”**

*Operand* # *Operand*

Returns the right Operation appended to the left Operand.

Concatenation returns a value that is the same type of the left Operand.

Ex: “this is” # “ a string” = “this is a string”

Ex: 100 # “0.333” = 1000.333

Ex: “222” # 4.04 = “2224.04”

Concatenation on its own supports the concatenation of between two different types, however assigning a concatenation to an identifier is treated differently.

Assigning Concatenations that are not supported:

Concatenation does not support the concatenation of two Float Operands.

Ex: Float f = 10.10 # 0.22 >> ERROR

Concatenation between a numeric operand and a string operand that results in a non numeric value results in an error.

Ex: Int i = 100 # “a string” >> ERROR

Ex: String s = “a string” # 100 >> “a string100”

Concatenation between a boolean operand and any other operand results in an error.

Ex: Bool b = T # “a string” >> ERROR

Ex: String s = “a string” # T >> “a stringT”

**Section 3.4 Operator Precedence**

Operators, Functions, and Arrays have precedence in HavaBol.

HavaBol’s precedence is described in the table below.

|  |  |
| --- | --- |
| ARRAYS | BUILT-IN FUNCTIONS | 16 |
| PARENTHESIS | 15 |
| UNARY MINUS | 12 |
| EXPONENTIATION | 11 |
| MULTIPLICATION | DIVISION | 9 |
| ADDITION | SUBTRACTION | 8 |
| CONCATENATION | 7 |
| CONDITIONAL OPERATIONS | 6 |
| LOGICAL NOT | 5 |
| LOGICAL AND | LOGICAL OR | 4 |

(See Section 3.3 for more information on these operators)

**Section 3.5 Assignment Operator “=”**

Assignment Operations in HavaBol must be in a declaration statement.

Using an assignment operator anywhere else will result in an error.

(for more information on Assignment, check out Chapter 1).

**Section 3.6 Numeric Assignment Operators “+=” “-=”**

**\*\*\*Numeric Assignment Operators are not supported in HavaBol**

**Chapter 4 Slices**

**\*\*\*Slices are not supported in HavaBol.**

**Chapter 5 Flow Control**

**Section 5.1 if statements**

If statements allow you the programmer to execute specific lines of code based off the value of a logical operation or a Boolean Identifier.

HavaBol supports optional else statements, as well as the nesting of if statements.

To create an if statement in HavaBol, follow this basic structure:

if *Condition* :

// code

else :

// more code

endif ;

NOTE:

1. if and else must be terminated by a “:” symbol.
2. if statements must be terminated by an endif (which is terminated by a “;” symbol).
3. *Condition* denotes an operation that results in a Boolean, or a value that is coercible to a Boolean.
   1. If the condition evaluates to true the if block will be executed and the else block will be ignored.
   2. If the condition evaluates to false then the if block will be ignored and the else block will be executed if there is one.
4. the else block of code is completely optional.

Here’s an example use of if statements that includes nested ifs:

if T:

print(“this is a true if statement”);

Int i = 12;

if i == 10:

print(“i is equal to 10”);

else:

print(“i isn’t equal to 10”);

endif;

endif;

print(“end of if statement”);

Output:

this is a true if statement

i isn’t equal to 10

end of if statement

**Section 5.2 while statements**

While statements allow you the programmer to execute a number of lines of code in a continuous loop until the conditional statement denoted evaluates to false.

HavaBol supports the nesting of while statements.

To create a while loop follow this basic structure:

while *Condition* :

// code goes here

endwhile;

NOTE:

1. while must be terminated by a “:” symbol.
2. while statements must be terminated by an endwhile (which is terminated by a “;” symbol.
3. *Condition* denotes an operation that results in a Boolean, or a value that is coercible to a Boolean.
   1. If the condition evaluates to true, the while block of code executes and checks the condition again.
   2. If the condition evaluates to false, the while block of code is skipped and the program continues executing.

Here’s an example of use of while statements that includes nested whiles:

Int i = 0;

Int j;

while i < 2:

print(“i = “, i);

j = 0;

while j < 3:

print(“j = “, j);

j = j + 2;

endwhile;

i = i + 1;

endwhile;

print(“end of while loop”);

Output:

i = 0

j = 0

j = 2

i = 1

j = 0

j = 2

end of while loop

Note that it is possible to create an infinite while loops in HavaBol. This is not recommended since the conditional argument is the only way to exit a while loop in HavaBol.

Example of an infinite while loop:

while T:

print(“this never stops”);

endwhile;

Output:

this never stops

this never stops

this never stops

this never stops

this never stops

this never stops

. . .

**Section 5.3 for loops**

In this section we will give a detailed overview as to how for loops work in HavaBol. For loops are a type of iteration similar to while loops. Except with for loops, the number of iterations is known to the programmer using the for loop. There are several ways the programmer knows the number of iterations the loop will iterate beforehand, which we will discuss in detail in this section. To begin, there are three different types of for loops in HavaBol:

1. Counting for loop
2. foreach loop
3. for loop with string tokenizer

Every for loop in HavaBol begins with the for keyword, followed by the parameters required respective to the type of for loop. The counting for loop in particular also has an optional parameter. Every for loop in HavaBol, regardless of the type of for loop, requires the following terminating strings:

1. A single colon “:” immediately proceeding the for loop condition.
2. The keyword “endfor” after the last statement of the loop.
3. A terminating semicolon “;” immediately proceeding the keyword “endfor”

Although HavaBol is whitespace insensitive, the following will result in a runtime error regarding the three requirements stated above:

1. Any non whitespace character proceeding a for loop condition that is not a colon “:”.
2. A missing “endfor” keyword after the last statement of the loop.
3. Any non whitespace character that is not a semicolon “;” immediately proceeding the “endfor” keyword.

These are not the only runtime errors that can occur in a for loop, they are simply the three errors that can occur in any for loop. We will discuss further runtime errors that may occur when we reach the respective for loop.

**Section 5.4 The Counting for Loop**

The counting for loop is used to iterate a set number of times as determined by the programmer. The counting for loop takes a minimum of two parameters, and a maximum of three. It also takes a required equal sign “=”, “to” keyword, and an optional “by” keyword. Lastly, it takes a required colon “:” to terminate the for loop condition. The specified structure of a counting for loop is as follows:

for *controlValue* = *startingValue* to *limit* by *increment*:

*statementsBody;*

endfor;

Where each token is defined as follows:

* 1. for – The keyword to instantiate a for loop. You cannot run a for loop without this keyword.
  2. *controlValue* – This is the value that will be incremented with each iteration of the for loop. By default, this value is incremented by 1, unless an increment value is specified. If an increment value is specified, *controlValue* will be incremented by the specified increment value. *controlValue* will hold its current value on each iteration of the loop, and can be used as so. However, if the programmer ever modifies this value within the for loop, such as adding to it, subtracting from it, or any arithmetic operation that modifies its value other than it was previously, it will impact the number of iterations the for loop will run. When the loop reaches the top again, *controlValue* will have the value that the programmer modified it to be, and then be incremented accordingly. The for loop will terminate when *controlValue* is equal to the limit specified by the programmer. The requirements of *controlValue* are as follows:
     1. *controlValue* must be a scalar identifier. It does not have to be declared nor initialized, but it must be a scalar identifier.
        1. Note: If previously declared, the identifier must be an integer identifier.

A *controlValue* that does not meet these requirements will result in a runtime error.

* 1. = - A single equal sign. Not having an equal sign “=” between the *controlValue* and *startingValue* will result in a runtime error.
  2. *startingValue* – The *startingValue* is the value that the *controlValue* will be set to on the first iteration of the for loop. After the initial iteration of the for loop, the *startingValue* is no longer used, needed, nor influential on the number of iterations. The *startingValue* must be one of the following:
     1. An integer constant
     2. A float constant.
        1. Note: If the *startingValue* is a float constant the decimal portion will be truncated. Example:
           1. 3.0 = 3
           2. 3.14 = 3
     3. An integer identifier
     4. A float identifier
        1. Note: A float identifier will result in the decimal being truncated as shown in the example of a float constant.
     5. A mathematical expression that results in a integer or float constant.
        1. Note: An expression that results in a float constant will result in the decimal portion being truncated.

A *startingValue* that does not meet one of these requirements will result in a runtime error.

* 1. to – The “to” keyword is required in between the *startingValue* and the limit. Not having the “to” keyword will result in a runtime error.
  2. *limit* – The *limit* is the value that will determine when the for loop stops iteration. The for loop will iterate until the *controlValue* is equal to the *limit*. Please note, modifying the *limit* value will not impact the number of iterations the loop will iterate. The *limit* value is permanent once the for loop begins its first iteration. The *limit* value must be one of the following:
     1. An integer constant
     2. A float constant.
        1. Note: If the *startingValue* is a float constant the decimal portion will be truncated. Example:
           1. 3.0 = 3
           2. 3.14 = 3
     3. An integer identifier
     4. A float identifier
        1. Note: A float identifier will result in the decimal being truncated as shown in the example of a float constant.
     5. A mathematical expression that results in a integer or float constant.
        1. Note: An expression that results in a float constant will result in the decimal portion being truncated.

A *limit* that does not meet one of these requirements will result in a runtime error.

* 1. by – The “by” keyword is optional, and only needed if an increment value is specified by the programmer. After parsing the *limit* value, HavaBol will look for either a “by” keyword or a colon “:”. If HavaBol finds any other character either than one of these, (not including whitespace) a runtime error will result. However, if a “by” keyword is found, the next value must be an *increment* value.
  2. *increment* – Although this value is optional, it is necessary if a “by” keyword is present. If a “by” keyword is specified and the next token is anything other than a valid *increment*, a runtime error will occur. Please note, modifying the *increment* value will not impact the number of iterations the loop will iterate. The *increment* value is permanent once the for loop begins its first iteration. A valid *increment* value must be one of the following:
     1. An integer constant
     2. A float constant.
        1. Note: If the *startingValue* is a float constant the decimal portion will be truncated. Example:
           1. 3.0 = 3
           2. 3.14 = 3
     3. An integer identifier
     4. A float identifier
        1. Note: A float identifier will result in the decimal being truncated as shown in the example of a float constant.
     5. A mathematical expression that results in a integer or float constant.
        1. Note: An expression that results in a float constant will result in the decimal portion being truncated.

An *increment* that does not meet one of these requirements will result in a runtime error.

Note: If an *increment* value is not provided, the for loop will use a default *increment* value of 1.

* 1. : - The colon “:” is necessary to terminate every for loop, whether nested or parent. Not having a colon immediately proceed a *limit* value or an *increment* value will result in a runtime error.
  2. endfor – The keyword used to end a for loop body.
  3. ; - A semicolon to immediately proceed the endfor keyword.

We will conclude this section with a few examples of valid for loops:

Example 1:

for i = 0 to 5:

print (i);

endfor;

Example 2:

for i = 0 to 5 by 2:

print (i);

endfor;

Example 3:

Int j = 2;

for i = 0 to j \* 5:

print (i);

endfor;

Example 4:

Int limit = 5;

Int j = 2;

for i = 0 to limit by j:

print (i);

endfor;

**Section 5.5 The foreach Loop**

The foreach loop is used to iterate over a set of values specified by the programmer. This loop will iterate one time for every value in the set. However, it will not iterate over a value if the value is uninitialized or null. In other words, if a value is uninitialized or null, it will skip the entire iteration. The foreach loop takes two required parameters, and no optional. It also takes a required “in” keyword and a required “:” colon to terminate the for loop. The structure for a foreach loop is as follows:

for *controlVariable* in *set*:

*statementsBody;*

endfor;

Where each token is defined as follows:

* 1. for – The keyword to instantiate a for loop. You cannot run a for loop without this keyword.
  2. *controlVariable* – This is the variable that will be assigned a new value in the set specified each iteration. It will have a new value on each iteration of the loop. Although this variable can be modified, it will not impact the number of iterations in the loop. The loop will still run for every initialized and non-null value in the set. When the for loop exits, this variable will have the last initialized/non-null value in the array. *controlVariable* must be one of the following requirements:
     1. An undeclared identifier.
     2. A declared identifier that is either:
        1. The same datatype of the set specified.
        2. A datatype that can be casted to the data type of the set specified. See **Chapter One: Data Types** for information on valid data type casting.

A *controlVariable* that is not one of these requirements will result in a runtime error.

Note: The *controlVariable* will be assigned a value according to the following criteria:

1. If the set is an array, it will be assigned each element of the array as long as the element is initialized/non-null.
2. If the set is a string identifier or string literal, it will be assigned each character of the string value.
   1. in – The “in” keyword is required. It is used to specify a foreach loop and is required for a foreach loop.
   2. *set* – The *set* value is the set of which the *controlVariable* will be assigned a value from every iteration of the loop. The *controlVariable* will be assigned a value from this set in sequential order as long as the present value is initialized/non-null. If any value is uninitialized/null, the *controlVariable* will skip the iteration of the loop and the entire statements body inside it. The set must meet one of the following requirements:
      1. An array identifier
      2. A string identifier
      3. A string literal

A set that does not meet one of these requirements, a runtime error will occur.

* 1. : - The colon “:” is necessary to terminate every for loop, whether nested or parent. Not having a colon immediately proceed a *limit* value or an *increment* value will result in a runtime error.
  2. endfor – The keyword used to end a for loop body.
  3. ; - A semicolon to immediately proceed the endfor keyword.

We will conclude this section with a few examples of valid foreach loops:

Example 1:

Int array[5] = 1, 2, 3;

for item in array:

print (item);

endfor;

Example 2:

Int array[5] = 1, 2, 3;

Int item;

for item in array:

print (item);

endfor;

Example 3:

String str = “this is a string”:

for ch in str:

print (ch);

endfor;

Example 4:

for ch in “this is a string”:

print (ch);

endfor;

**Section 5.6 The for tokenizer Loop**

The for tokenizer loop is a loop is given both a string, and a delimiter to split that string on. It then splits the string based on the delimiter provided, and iterates over the tokens resulting from the split. When the string is split, it results in an array where each element of the array is the string leading up to the delimiter but excluding the delimiter. The structure of the for tokenizer loop is as follows:

for *stringControlVariable* from *String* by *delimiter*:

*statementsBody;*

endfor;

Where each token is defined as follows:

1. for – The keyword to instantiate a for loop. You cannot run a for loop without this keyword.
2. *stringControlVariable* - This is the variable that will be assigned a new token resulting from the string split on each iteration. It will have a new value on each iteration of the loop. Although this variable can be modified, it will not impact the number of iterations in the loop. The loop will still run for element resulting from the string split. When the for loop exits, this variable will be the last element resulting from the token split. *controlVariable* must be one of the following requirements:
   * 1. An undeclared identifier.
     2. A declared string identifier. The declared string identifier does not have to be initialized.

A *stringControlVariable* that does not meet one of these requirements will result in a runtime error.

1. from – The “from” keyword is used to specify a string tokenizer for loop. It is necessary for this type of for loop.
2. *String* – *String* is the parent string that will be split on by the delimiter. This string will be split on the delimiter provided, and an array will result, which *stringControlVariable* will then iterate over each element. Although this value can be modified throughout each iteration, it does not impact the number of iterations of the loop. The *String* must be one of the following:
   1. A string literal
   2. A string identifier

A *String* that is not one of these requirements will result in a runtime error.

1. by – The “by” keyword is necessary for the string tokenizer loop and is used to specify the delimiter for the *String* to be split on.
2. *delimiter* – This is the variable that the programmer will specify to split the *String* on. Each element of the resulting array (that the programmer cannot access) will be the part of the string before each *delimiter* provided, excluding the *delimiter*. The *delimiter* must be one of the following:
   * 1. A string literal
     2. A string identifier

A *delimiter* that does not meet one of these requirements will result in a runtime error.

We will conclude this section with a few examples of valid for tokenizer loops:

Example 1:

String str = “string, comma, delimited”;

String delimiter = “,”;

for token from str by delimiter:

print (token);

endfor;

Example 2:

String token;

for token from “string, comma, delimited” by “,”:

print (token);

endfor;

**Section 5.7 Select/When Statements**

**\*\*\*Select/When statements are not supported in HavaBol**

**Section 5.8 Break/Continue Statements**

**\*\*\*Break/Continue statements are not supported in HavaBol**

**Chapter 6 Built-In Functions**

**Section 6.1 Print**

The print function allows you the programmer to output values to the screen.

The print function is implemented as follows:

print ( *Operand*, *Operand*, *Operand* ... );

NOTE:

1. The print function allows for a variable number of Operands
2. Operands can be identifiers
3. Operands can be of primitive data types
4. Operands can be of Arrays
5. Operands can be Array/String References
6. Operands can be standalone expressions
7. Operands must be separated by commas “,”
8. Operands are appended to each other when printed out (not separated by spaces)
9. print always appends a newline (“\n”) character to the printed out message

Example:

print(“this ”, “is” # “ “, “a weird string”, “ with the #”

, 2 \* 4 / 1 ^ 3);

Int i = 10;

Float f = 1222.2;

Bool b = T;

print(i, “ “, f, “ “, b);

Output:

this is a weird string with the #512

10 1222.2 T

**Section 6.2 LENGTH**

The LENGTH function takes in an Operand and returns how many characters it is.

The LENGTH function is implemented as follows:

LENGTH ( *Operand* );

*Operand* can be of any data type, with the exception of Arrays.

The function keyword LENGTH must be in all caps.

Example:

String s = “this is 20 characters”;

print(LENGTH(s));

Output:

20

**Section 6.3 SPACES**

The SPACES function takes in an Operand and returns true if the Operand is empty or if the Operand is nothing but spaces, and returns false otherwise.

The SPACES function is implemented as follows:

SPACES ( *Operand* );

*Operand* can be of any data type, with the exception of Arrays.

The function keyword SPACES must be in all caps.

Example:

String s = “”;

String s2 = “ “;

String s3 = “this has stuff in it”;

print(SPACES(s));

print(SPACES(s2));

print(SPACES(s3));

Output:

T

T

F

**Section 6.4 ELEM**

The ELEM function takes in an Array Operand and returns the index of the last initialized element + 1.

The ELEM function is implemented as follows:

ELEM ( *Operand* );

Operand must be an Array. ELEM does not support other primitive data types or Array element references.

The function keyword ELEM must be in all caps.

Example:  
 Int i[10] = 1, 2, 3, 4, 5;

print(ELEM(i));

Output:

5

**Section 6.5 MAXELEM**

The MAXELEM function takes in an Array Operand and returns differently based off of whether the Array is bounded or unbounded.

If the Array Operand is a bounded Array MAXELEM will return the maximum amount of elements that can be stored in the passed Array.

If the Array Operand is an unbounded Array MAXELEM will return the index of the highest initialized element + 1.

The MAXELEM function is implemented as follows:

MAXELEM ( *Operand* );

Operand must be an Array. MAXELEM does not support other primitive data types or Array element references.

The function keyword MAXELEM must be in all caps.

Example:

Int i[10] = 1, 2, 3, 4, 5;

Int i2[unbound] = 1, 2, 3, 4, 5;

print(MAXELEM(i));

print(MAXELEM(i2));

Output:

10

5

**Section 6.6 Date Functions**

**\*\*\*dateDiff, dateAdj, dateAge are not supported by HavaBol**

**Chapter 7 Programmer-Defined functions**

**\*\*\*Programmer-Defined functions are not supported in HavaBol.**

**Chapter 8 Test Cases**

8.1 Counting For Loop Valid Test Cases

The file titled “for\_loop.txt” is the test file that contains valid for loop test cases. The test cases in this file range from simple to complex. The valid for loop cases this file tests are the following:

1. Counting for loops (including nested and unnested)::
   1. Integer constant as both the startingValue and limit, no increment value.
   2. Integer constant as the startingValue and limit, with an integer identifier as the increment.
   3. Builtin functions such as ELEM(array) as the startingValue, with an integer identfiier as the limit, and integer constant as the increment value.
   4. Integer constant as the starting value, limit as an integer identifier, integer constant as the increment.
   5. Builtin functions as the startingValue, limit, and increment.
   6. Integer constant as the controlVariable, builtin function as the limit, no increment.
2. Foreach loops (including nested and unnested):
   1. Undeclared identifier as the controlVariable, integer array as the set to iterate over.
   2. Undeclared identifier as the controlVariable, string identifier as the set to iterate over.
   3. Declared identifer as the controlVariable, string literal as the set to iterate over.
   4. Undeclared identifier as the controlVariable, string identifier as the set to iterate over.
3. For tokenizer loops (including nested and unnested):
   1. Undeclared identifier as the stringControlVariable, string identfier as the String set, string identifier as the delimiter.
   2. Declared identifier as the stringControlVariable, string literal as the String set, string literal as the delimiter.
   3. Declared identifier stringControlVariable, string identifer as the String set, string literal as the delimiter.

Left off on this loop in for\_loop.txt:

for ch from szStr by ",":

print (ch);

for ch from "for! testing!" by "!":

print (ch);

endfor;

endfor;

8.2 Counting For Loop Error Test Cases

|  |  |
| --- | --- |
| incrIdentifierNotInteger | Increment variable is not an integer. |
| incrInvalid | Increment is a string literal and thus invalid. |
| incrNoValue | Increment is an uninitialized identifier integer identifier. |
| incrUndeclared | Increment variable is undeclared. |
| limitIdentifierNotInteger | Limit variable is a string. |
| limitInvalid | Limit is a string literal. |
| limitNoValue | Limit is an uninitialied integer identifier. |
| limitUndeclared | Limit is an undeclared identifier. |
| missingByKeyword | Missing “by” keyword between the limit and increment. |
| missingColonWithBy | Has increment but missing colon “:”. |
| missingColonWithoutBy | No “by” keyword and no colon “:”. |
| missingEndfor | For loop is missing “endfor” token after loop. |
| missingEndforSemicolon | “endfor” is missing a terminating semicolon “;”. |
| missingEqualSign | Missing equal sign “=” between controlVariable and startingValue. |
| missingToKeyword | Missing “to” keyword between startingValue and limit. |
| startIdentifierNotInteger | startingValue is a string identifier. |
| startInvalid | startingValue is a string literal. |
| startNoValue | startingValue is an uninitialized integer identifier. |
| controlNotIdentifier | controlVariable is an integer constant. |
| startUndeclared | startingValue is an undeclared identifier. |

8.3 Nested Counting For Loop Error Test Cases

|  |  |
| --- | --- |
| incrIdentifierNotInteger | Increment variable is not an integer. |
| incrInvalid | Increment is a string literal and thus invalid. |
| incrNoValue | Increment is an uninitialized identifier integer identifier. |
| incrUndeclared | Increment variable is undeclared. |
| limitIdentifierNotInteger | Limit variable is a string. |
| limitInvalid | Limit is a string literal. |
| limitNoValue | Limit is an uninitialied integer identifier. |
| limitUndeclared | Limit is an undeclared identifier. |
| missingByKeyword | Missing “by” keyword between the limit and increment. |
| missingColonWithBy | Has increment but missing colon “:”. |
| missingColonWithoutBy | No “by” keyword and no colon “:”. |
| missingEndfor | For loop is missing “endfor” token after loop. |
| missingEndforSemicolon | “endfor” is missing a terminating semicolon “;”. |
| missingEqualSign | Missing equal sign “=” between controlVariable and startingValue. |
| missingToKeyword | Missing “to” keyword between startingValue and limit. |
| startIdentifierNotInteger | startingValue is a string identifier. |
| startInvalid | startingValue is a string literal. |
| startNoValue | startingValue is an uninitialized integer identifier. |
| controlNotIdentifier | controlVariable is an integer constant. |
| startUndeclared | startingValue is an undeclared identifier. |

8.4 Foreach Loop Error Test Cases

|  |  |
| --- | --- |
| arrayUndeclared | Array is an undeclared identifier. |
| controlBoolIntArray | controlVariable is a boolean identifier, array is an integer identifier. |
| controlBoolStringArray | controlVariable is a boolean identifier, array is a string array. |
| controlBoolStringLiteral | controlVariable is a boolean identifier, set is a string literal. |
| controlFloatStringArray | controlVariable is a float identifier, array is a string array. |
| controlFloatStringLiteral | controlVariable is a float identifier, set is a string literal. |
| controlIntegerStringArray | controlVariable is an integer identifier, array is a string array. |
| controlIntegerStringLiteral | controlVariable is an integer identifier, set is a string literal. |
| controlNotIdentifier | controlVariable is an integer constant. |
| missingColon | Loop condition is missing a terminating colon “:”. |
| missingEndfor | Loop is missing a terminating “endfor” keyword after the loop. |
| missingEndforSemicolon | “endfor” is missing a terminating semicolon “;”. |

8.5 Nested Foreach Loop Test Cases

|  |  |
| --- | --- |
| arrayUndeclared | Array is an undeclared identifier. |
| controlBoolIntArray | controlVariable is a boolean identifier, array is an integer identifier. |
| controlBoolStringArray | controlVariable is a boolean identifier, array is a string array. |
| controlBoolStringLiteral | controlVariable is a boolean identifier, set is a string literal. |
| controlFloatStringArray | controlVariable is a float identifier, array is a string array. |
| controlFloatStringLiteral | controlVariable is a float identifier, set is a string literal. |
| controlIntegerStringArray | controlVariable is an integer identifier, array is a string array. |
| controlIntegerStringLiteral | controlVariable is an integer identifier, set is a string literal. |
| controlNotIdentifier | controlVariable is an integer constant. |
| missingColon | Loop condition is missing a terminating colon “:”. |
| missingEndfor | Loop is missing a terminating “endfor” keyword after the loop. |
| missingEndforSemicolon | “endfor” is missing a terminating semicolon “;”. |

8.6 For Tokenizer Loop Test Cases

|  |  |
| --- | --- |
| delimiterNoValue | delimiter is an uninitialized string identifier. |
| delimiterNotString | delimiter is an integer identifier. |
| delimiterNotString2 | delimiter is an integer constant. |
| delimiterUndeclared | delimiter is an undeclared identifier. |
| iterableNotIdentifier | stringControlVariable is an integer constant. |
| iterableNotStringIdentifier | stringControlVariable is an integer identifier. |
| missingColon | Loop condition is missing a terminating colon “:”. |
| missingEndfor | Missing “endfor” keyword after the loop. |
| missingEndforSemicolon | “endfor” is missing a terminating semicolon “;”. |
| stringNoValue | String is uninitialized. |
| stringNotAString | String is an integer identifier. |
| stringNotAString2 | String is an integer constant. |
| stringUndeclared | String is an undeclared identifier. |

8.7 Nested For Tokenizer Loop Test Cases

|  |  |
| --- | --- |
| delimiterNoValue | delimiter is an uninitialized string identifier. |
| delimiterNotString | delimiter is an integer identifier. |
| delimiterNotString2 | delimiter is an integer constant. |
| delimiterUndeclared | delimiter is an undeclared identifier. |
| iterableNotIdentifier | stringControlVariable is an integer constant. |
| iterableNotStringIdentifier | stringControlVariable is an integer identifier. |
| missingColon | Loop condition is missing a terminating colon “:”. |
| missingEndfor | Missing “endfor” keyword after the loop. |
| missingEndforSemicolon | “endfor” is missing a terminating semicolon “;”. |
| stringNoValue | String is uninitialized. |
| stringNotAString | String is an integer identifier. |
| stringNotAString2 | String is an integer constant. |
| stringUndeclared | String is an undeclared identifier. |

8.8 Assignment Valid Test Cases

The file titled “goodDeclare.hav” is the test file that contains valid assignment test cases. The test cases in this file range from simple to complex. The valid assignment cases this file tests are the following:

1. Simple integer, float, String, boolean assignment.
2. Simple array declaration and assignment. Examples with all datatypes.
3. Simple string assignment concatenation.
4. Integer assignment using a concatenation with an integer string and integer. Integer string is parsed into its integer value then the trailing integer is concatenated.
5. Float assignment using concatenation with floats as the 2 operands.
6. Simple assignment of a specific array index. Examples with all datatypes.
7. Boolean array/declaration with a boolean expression as one element.

8.9 Array Assignment Error Test Cases

|  |  |
| --- | --- |
| arrayDefault.hav | Assigning an integer array to a string literal. |
| arrayDefault2.hav | Assigning an integer array to a boolean variable. |
| arrayDefault3.hav | Assigning a float array to a string literal. |
| arrayDefault4.hav | Assigning a float array to a boolean variable. |
| arrayDefault5.hav | Assigning a boolean array to an integer variable. |
| arrayDefault6.hav | Assigning a boolean array to a float variable. |
| arrayDefault7.hav | Assigning a boolean array to a string variable. |
| badArrayItemDeclare.hav | Declaring an integer array with a boolean element. |
| badArrayItemDeclare2.hav | Declaring an integer array with a string literal element. |
| badArrayItemDeclare3.hav | Declaring a float array with a boolean element. |
| badArrayItemDeclare4.hav | Declaring a float array with a string literal element. |
| badArrayItemDeclare5.hav | Declaring a boolean array with a float element. |
| badArrayItemDeclare6.hav | Declaring a boolean array with a float element. |
| badArrayItemDeclare7.hav | Declaring a boolean array with a string literal element. |
| badArrayItemDeclareFixed.hav | Declaring a fixed size integer array with a boolean element. |
| badArrayItemDeclareFixed2.hav | Declaring a fixed size integer array with a string literal element. |
| badArrayItemDeclareFixed3.hav | Declaring a fixed size float array with a boolean element. |
| badArrayItemDeclareFixed4.hav | Declaring a fixed size float array with a string literal element. |
| badArrayItemDeclareFixed5.hav | Declaring a fixed size boolean array with a float element. |
| badArrayItemDeclareFixed6.hav | Declaring a fixed size boolean array with a float element. |
| badArrayItemDeclareFixed7.hav | Declaring a fixed size boolean array with a string literal element. |
| badArrayItemDeclare\_unbound.hav | Declaring an unbounded integer array with a boolean element. |
| badArrayItemDeclare\_unbound2.hav | Declaring an unbound integer array with a string literal element. |
| badArrayItemDeclare\_unbound3.hav | Declaring an unbound float array with a boolean element. |
| badArrayItemDeclare\_unbound4.hav | Declaring an unbound float array with a string literal element. |
| badArrayItemDeclare\_unbound5.hav | Declaring an unbound boolean array with a float element. |
| badArrayItemDeclare\_unbound6.hav | Declaring an unbound boolean array with a float element. |
| badArrayItemDeclare\_unbound7.hav | Declaring an unbound boolean array with a string literal. |
| copyArray.hav | Copying a string array to an integer array. |
| copyArray2.hav | Copying a boolean array to an integer array. |
| copyArray3.hav | Copying a string array to a float array. |
| copyArray4.hav | Copying a boolean array to a float array. |
| copyArray5.hav | Copying an integer array to a boolean array. |
| copyArray6.hav | Copying a float array to a boolean array. |
| copyArray7.hav | Copying an unbound string array to an unbound boolean array. |
| invalidArrdec.hav | Declared array size is a string literal. |
| invalidArraydec2.hav | Declared array size is a boolean. |

8.10 Assignment Error Test Cases

|  |  |
| --- | --- |
| invalidCast1.hav | Casting boolean to integer. |
| invalidCast2.hav | Casting string to integer. |
| invalidCast3.hav | Casting string to float. |
| invalidCast4.hav | Casting boolean to float. |
| invalidCast5.hav | Casting integer to boolean. |
| invalidCast6.hav | Casting float to boolean. |
| invalidCast7.hav | Casting string to boolean |
| forgotQuotes.hav | Printing an unidentified identifier/string without quotes. |
| invalidDec.hav | Declaring integer with “int” instead of proper “Int” keyword. |
| invalidDec2.hav | Question mark “?” terminating declaration instead of semicolon “;”. |
| invalidDec3.hav | Missing semicolon “;” on integer declaration. |
| missingSem.hav | Missing semicolon “;” in print statement. |

8.11 Built-In Functions Valid Test Cases

The file titled “goodFuncs.hav” is the test file that contains valid built-in function test cases. The valid built-in test cases this file tests are the following:

1. ELEM(array):
   1. Inside a print statement.
   2. Integer variable assignment.
2. MAXELEM(array):
   1. Inside a print statement.
   2. Integer variable assignment.
3. LENGTH(str):
   1. Inside a print statement.
   2. Integer variable assignment.
4. SPACES(str):
   1. Inside a print statement.
   2. Boolean variable assignment.

8.12 Built-In Functions Error Test Cases

|  |  |
| --- | --- |
| boolToNum.hav | Print statement comparing a boolean to an integer. |
| elemOnBool.hav | Calling ELEM() on a boolean. |
| elemOnFloat.hav | Calling ELEM() on a float. |
| elemOnInt.hav | Calling ELEM() on an integer. |
| elemOnString.hav | Calling ELEM() on a string. |
| lengthOpIsArray.hav | Calling LENGTH() with an array. |
| maxElemOpNotArray.hav | Calling MAXELEM() on a string. |
| spacesOpIsArray.hav | Calling SPACES() with an array. |
| missingOperands.hav | Calling LENGTH() with no operands. |
| operandNotCreated.hav | Calling ELEM() with an undeclared array. |

8.13 Expression Valid Test Cases

The file titled “goodExprCases.hav” is the test file that contains valid expr test cases. The valid expr test cases this file tests are the following:

1. A complex integer expression assignment involving 4 operands and 3 operators.
2. A complex float expression assignment involving 7 integer operands and 6 operators.
3. Boolean assignment involving 3 operands and 2 operators.
4. A complex integer assignment involving 8 operands including 2 float operands.

8.14 Expression Error Test Cases

|  |  |
| --- | --- |
| missingLParen | Expression missing left parentheses. |
| missingRParen | Expression missing right parentheses. |
| arrayAsLOperand | Left operand is an array without an index, right operand is an array with an index. |
| arrayAsROperand | Left operand is an array with an index, right operand is an array without an index. |
| arrayIndexNegNumeric | Array index is a large negative number. |
| arrayIndexNonNumeric | Array index is a string identifier. |
| arrayMissingIndex | Array has brackets with no index. |
| arrayMissingIndexRefInOp | Array has index but missing brackets. |
| arrayMissingLBracket | Array being indexed has no index and missing right bracket. |
| arrayMissingRBracket | Array being indexed has no index and missing left bracket. |
| stringIndexNotNumeric | String index is a string identifier. |
| missingOperands | Expression is missing left and right operands. |
| missingOperator | Expression is missing operator. |
| operandNotCreated | Expression has an undeclared identifier. |
| andLOpNotBool | And comparison, left operand is an integer. |
| andROpNotBool | And comparison, right operand is an integer. |
| equalLOpBool | Equal comparison, left operand is a boolean. |
| equalNotLOpBool | Equal comparison, left operand is not a boolean. |
| equalNotROpBool | Equal comparison, right operand is not a boolean. |
| equalROpBool | Equal comparison, right operand is a boolean. |
| greatEqLOpBool | Greater than equal to comparison, left operand is a boolean. |
| greatEqROpBool | Greater than equal to comparison, right operand is a boolean. |
| greatLOpBool | Greater than comparison, left operand is a boolean. |
| greatROpBool | Greater than comparison, right operand is a boolean. |
| lessEqLOpBool | Less than equal to comparison, left operand is a boolean. |
| lessEqROpBool | Less than equal to comparison, right operand is a boolean. |
| lessLOpBool | Less than comparison, left operand is a boolean. |
| lessROpBool | Less than comparison, right operand is a boolean. |
| notOpNotBool | Not operation, operand is not a boolean. |
| orLOpNotBool | Or operation, left operand is not a boolean. |
| orROpNotBool | Or operation, right operand is not a boolean. |
| addLOpNonNum | Add operation, left operand is a string identifier. |
| addROpNonNum | Add operation, right operand is a string identifier. |
| divLOpNonNum | Division operation, left operand is a string identifier. |
| divROpNonNum | Division operation, right operand is a string identifier. |
| expoLOpNonNum | Exponentiation, left operand is a string identifier. |
| expoROpNonNum | Exponentiation, right operand is a string identifier. |
| mulLOpNonNum | Multiplication, left operand is a string identifier. |
| mulROpNonNum | Multiplication, right operand is a string identifier. |
| negOpNonNum | Negation operation, operand is a string identifier. |
| subLOpNonNum | Subtraction, left operand is a string identifier. |
| subROpNonNum | Subtraction, right operand is as string identifier. |

8.15 If Statement Valid Test Cases

The file titled “ifGoodCases.hav” is the test file that contains valid if statement test cases. The valid if statement test cases this file tests are the following:

1. A triple nested if else statement.
2. Each nested if statement also has an else block.
3. The if conditions evaluated are:
   1. Three integer comparions in one evaluation.
   2. Three integer comparisons involving three array elements in one evaluation.
   3. A string identifier to string literal comparison.
   4. A boolean comparison.

8.16 If Statement Error Test Cases

|  |  |
| --- | --- |
| elseMisspelled | “else” keyword is misspelled. |
| elseNotCorrectlyTerminated | “else” keyword is terminated with a semicolon “;”. |
| elseNotTerminated | “else” keyword missing terminating colon “:”. |
| endIfMisspelled | “endif” keyword is misspelled. |
| endIfNotCorrectlyTerminated | “endif” keyword is terminated with a colon “:”. |
| endIfNotTerminated | “endif” keyword missing terminating semicolon “;”. |
| ifNotCorrectlyTerminated | “if” keyword terminated with a semicolon “;”. |
| ifNotTerminated | “if” keyword missing terminating colon “:”. |
| noEndif | “endif” keyword is missing. |
| nonBooleanCondition | if condition is not a boolean. |

8.17 Nested If In Else Error Test Cases

|  |  |
| --- | --- |
| elseMisspelled | “else” keyword is misspelled. |
| elseNotCorrectlyTerminated | “else” keyword is terminated with a semicolon “;”. |
| elseNotTerminated | “else” keyword missing terminating colon “:”. |
| endIfMisspelled | “endif” keyword is misspelled. |
| endIfNotCorrectlyTerminated | “endif” keyword is terminated with a colon “:”. |
| endIfNotTerminated | “endif” keyword missing terminating semicolon “;”. |
| ifNotCorrectlyTerminated | “if” keyword terminated with a semicolon “;”. |
| ifNotTerminated | “if” keyword missing terminating colon “:”. |
| noEndif | “endif” keyword is missing. |
| nonBooleanCondition | if condition is not a boolean. |

8.18 While Statement Valid Test Cases

The file titled “whileGoodCases.hav” is the test file that contains valid while statement test cases. The valid while statement test cases this file tests are the following:

1. A tripled nested while loop.
2. The three conditions evaluated in each while loop are the following:
   1. An integer comparison in an evaluation.
   2. Two integer comparions in one evaluation.
   3. A boolean comparison.

8.19 While Statement Error Test Cases

|  |  |
| --- | --- |
| endWhileNotCorrectlyTerminated | “endwhile” keyword terminated with a colon “:”. |
| endWhileNotTerminated | “endwhile” keyword missing terminating semicolon “;”. |
| noEndWhile | “endwhile” keyword is missing. |
| nonBooleanCondition | while condition is not a boolean. |
| whileNotCorrectlyTerminated | “while” keyword terminated with a semicolon “;”. |
| whileNotTerminated | “while” keyword missing terminating colon “:”. |

8.20 Nested While Statement Error Test Cases

|  |  |
| --- | --- |
| innerEndWhileNotCorrectlyTerminated | Inner “endwhile” keyword terminated with a colon “:”. |
| innerEndWhileNotTerminated | Inner “endwhile” keyword missing terminating semicolon “;”. |
| innerNonBooleanCondition | Inner while condition is not a boolean. |
| innerWhileNotCorrectlyTerminated | Inner “while” keyword terminated with a semicolon “;”. |
| innerWhileNotTerminated | Inner “while” keyword missing terminating semicolon “;”. |
| noInnerEndWhile | Inner “while” keyword missing terminating “endwhile” keyword. |