Sections

Introduction

Getting started

Chapter 1.) Data types

1.1 Int

1.2 Float

1.3 String

1.4 Bool

\*\* Examples should be included on each subsection

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 what constitutes an expression?

3.2 Operators

3.3 precedence

3.4 assignment??

Chapter 4 : Slices

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

5.1 if

5.2 while

5.3 for

Chapter 6 Builtin Functions

Flexible Points Completed

You must provide your own test data for

* unbounded arrays
* additional numeric assignment operators
* IN, NOTIN
* slices
* for tokenizer
* variable number of arguments to a function
* recursive use of a function
* integration between different capabilities (e.g., slices with IN, slices as arguments, slices as function result, unbound arrays as arguments, break/continue in select)

**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

**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

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 4 Slices**

**Slices were not implemented in HavaBol**

**Chapter 5 Flow Control**

**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.

**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;

**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 datatype 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;

**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;