PickleBOL Implementation

Two and a Half Caleb's – Caleb Scott, Caleb Evans and Luke DeGoes

1 Introduction

This document outlines our team's implementation of the pickle programming language. Each section includes a description of our implementation, important decisions we made designing the implementation, examples of pickle code, and information on included test files. Output from example code is commented in line with each print statement to describe the expected output in order to save space in the document.

2 Data Types

2.1 Numeric

There are two numeric data types: Int and Float. Both data types are implemented via a single Numeric Class. The Numeric Class contains attributes to store the String value, Integer value, Float value and Sub-Classification of the Numeric Object. The Class also contains methods to create, parse, and print Numeric objects.

Int and Float data types are outlined in the following sections.

2.1.1 Int

The Int data type represents an integer value as a four-byte integer.

As outlined in Section 2.1, our implementation of the Int data type uses a Numeric Class to handle creating, parsing, and printing Int objects. Both positive and negative integers are supported. Integers are parsed from Strings by utilizing the built in Java functions of the String Class (String.matches()).

An Int value will be coerced into a String or Float value as needed.

The Int data type can be used in conjunction with the following operations as defined in Section 4: addition, subtraction, multiplication, division, exponentiation, equal to, not equal to, less than, greater than, less than or equal to and greater than or equal to.

Example Code:

```
Int radius;
radius = 8;
Int radius2 = radius ^ 2;
print(radius);  // 8
print(radius2); // 64
```

Test Cases: Int Test Group

2.1.2 Float

The Float data type represents a floating-point value as an eight-byte floating-point (double).

As outlined in Section 2.1, our implementation of the Float data type uses a Numeric Class to handle creating, parsing, and printing Float objects. Both positive and negative floating-point values are supported. Floating-point values are parsed from Strings by utilizing the built in Java functions of the String Class (String.matches()).

Float objects are always printed with a minimum of two decimal places even though they may contain a less precise value. A Float value can be assigned to an Int and the Float value will be truncated during the assignment.

A Float value will be coerced into a String or Int value as needed.

The Float data type can be used in conjunction with the following operations as defined in Section 4: addition, subtraction, multiplication, division, exponentiation, equal to, not equal to, less than, greater than, less than or equal to and greater than or equal to.

Example Code:

```
Float pi = 3.14;
Float area;
Float radius2;
Int radius = 8;
Int truncated = pi;
radius2 = radius ^ 2;
area = pi * radius2;
print(pi);        // 3.14
print(area);      // 200.96
print(truncated); // 3
```

Test Cases: Float Test Group

2.2 Bool

The Bool data type represents a Boolean value as a single byte character 'T' or 'F'.

Our implementation of the Bool data type is done via a Bool Class. The Bool Class contains attributes to store the Boolean value, String value, Character value and Sub-Classification of the Bool object. The Class also contains methods to create, parse, and print Bool objects. Boolean values are parsed from Strings representing the single characters 'T' or 'F'.

A Bool value will be coerced into a String automatically if required (e.g. print() or LENGTH() built-in functions).

Example Code:

```
Bool bFlag;
bFlag = T;
print(bFlag); // T
```

Test Cases: Bool Test Group

2.3 String

The String data type represents a string of characters. String objects are mutable, and each character is a single byte.

Our implementation of the String data type is done via a ResultValue Class. The ResultValue Class contains attributes to store the String value and Sub-Classification of the String object. We did not implement an additional String Class because the existing ResultValue Class contained all the necessary parts to implement the String data type.

The String data type can be used in conjunction with the concatenation, subscripting, and generic operations as defined in Section 4. The data type can also be used with the built-in functions LENGTH and SPACES as defined in Section 7.

A String value can be coerced into a Numeric, Bool or Date value if it is of the correct format.

Slicing of String objects is supported as defined in Section 5.

Example Code:

```
String str;
str = "My name";
print(str);    // My Name
print("Jeff"); // Jeff
```

Test Cases: String Test Group

2.4 Date

The Date data type represents a date as a string of 10 characters in the YYYY-MM-DD format.

Our implementation of the Date data type is supported via a Date Class. The Date Class contains static attributes and methods to parse dates, convert Julian date format strings into YYYY-MM-DD strings, calculate differences in days, adjust dates by day counts and calculate differences in years. Calculations on dates is done via conversion of the YYYY-MM-DD date String into Julian date format integer to simplify calculations.

The Date data type is implemented as a String. Because Dates are implemented as strings, they support all the same operations of the String data type as defined in Section 2.3, and will be treated as Strings in any context outside of one of the built-in date functions defined below.

The Date data type is extended by the functions dateDiff, dateAdj and dateAge. Each are defined in Section 7.

Example Code:

```
Date date = "1999-02-27";
print(date);                                  // 1999-02-27
print(dateDiff("2021-02-27", date), "days"); // 8036 days
print(dateAge("2021-02-27", date), "years"); // 22 years
print(dateAdj(date, 10));                     // 1999-03-09
```

Test Cases: Date Test Group

3 Arrays

Arrays in pickles are homogeneous lists of the Int, Float, Bool or String data types. They can be either bounded or unbounded and can be declared and assigned in multiple ways. Arrays are indexed starting from 0, where 0 is the first element of the array.

Our implementation of arrays is done via a ResultList Class. The ResultList Class contains attributes to store the list of objects, the capacity of the array, the allocated size of the array and whether the array is unbounded or bounded.

A bounded array has a set capacity, which limits the maximum number of elements that can be present in the list. An array's allocated size is the index of the lists' highest populated element plus one. An unbounded array has a dynamic capacity, where the maximum number of elements increases/decreases based on the allocated size of the array.

An array in pickle is defined using the '[]' characters after the variable name of a variable declaration. Arrays can be unbounded (dynamically grow and shrink in size). An array can be defined in the following ways:

DataType name[] = *value1*, *value2*, *value3, …, valueN*;	Declare an Array of type *DataType*. Initialize the Array with *values*.
DataType name[*N*];	Declare an Array of type *DataType* with *N* capacity.
DataType name[*N*] = *value*;	Declare an Array of type *DataType* with *N* capacity, where all *N* values are *value*. (Scalar Assignment)
DataType name[*N*] = *value1*, *value2*, *value3, …, valueN*;	Declare an Array of type *DataType* with *N* capacity. Initialize the Array with *values*.
DataType name[unbound];	Declare an unbounded Array of type *DataType*.
DataType name[unbound] = *value1*, *value2*, *value3, …, valueN*;	Declare an unbounded Array of type *DataType*. Initialize the Array with *values*.

Arrays in pickle can be used in conjunction with the subscripting operation (see Section 4.4.1) , the built-in functions print, ELEM and MAXELEM (see Section 7.1) and slicing operations (see Section 5.2).

3.1 Array-to-Array Assignment

Arrays support array-to-array assignment, where an array can be assigned into another array. Array-to-Array assignment is based on the following rules:

- If a larger array is assigned into a smaller array, the larger array is copied into the smaller array until the smaller array is full.
- If a smaller array is assigned into a larger array, the smaller array is copied into the larger array until the smaller array is out of elements, then the rest of the larger array is filled with empty values up to its capacity.
- If an array is assigned into an array of the same size (or into an unbounded array) all elements will be copied to the target array.

Example Code: Int Test Group, Float Test Group, Bool Test Group, String Test Group, Date Test Group

```
// array with 4 elements
Int iArray1[] = 10, 20, 30, 40;
print(iArray1); // 10 20 30 40

// array of max size 3
Int iArray2[3];
iArray2[0] = 30;
iArray2[1] = 20;
iArray2[2] = 10;
print(iArray2); // 30 20 10

// scalar assignment of array with max size 6
Int iArray3[6] = 3;
print(iArray3); // 3 3 3 3 3 3

// Assigning larger array into smaller array
iArray2 = iArray1;
print(iArray2); // 10 20 30

// Assigning smaller array into larger array
iArray3 = iArray1;
print(iArray3); // 10 20 30 40

// Unbounded Array with 3 elements
Int unboundedArray1[unbound] = 20, 30, 40;
print(unboundedArray1);

// Unbounded Array declaration
Int unboundedArray2[unbound];
unboundedArray2[6] = 60;
print(unboundedArray2);
```

Test Cases:

4 Operations

4.1 Generic Operations

The behavior of generic operations is dependent on the left operand. If the left operand is a String, then the entire operation is treated as a String comparison, and the second operand will be coerced into a String if necessary. If the left operand is a Numeric the operation is treated as a Numeric comparison and the second operand must be a Numeric, else an error will occur.

4.1.1 Equal To (==)

The equal to operation returns a Bool object. The returned Bool object will be 'T' if both operands are equal, otherwise 'F' is returned.

The equal to operation is implemented via the Utility Class as a static method.

Example Code:

```
print("1" == 1);       // T
print("fun" == "fun"); // T
print("fun" == "nuf"); // F
print(1 == 1);         // T
print(1 == 2);         // F
```

Test Cases: Int Test Group, Float Test Group, Bool Test Group, String Test Group, Date Test Group

4.1.2 Not Equal To (!=)

The not equal to operation returns a Bool object. The returned Bool object will be 'T' if both operands are not equal, otherwise 'F' is returned.

The not equal to operation is implemented via the Utility Class as a static method.

Example Code:

```
print("1" != 1);       // F
print("fun" != "fun"); // F
print("fun" != "nuf"); // T
print(1 != 1);         // F
print(1 != 2);         // T
```

Test Cases: Int Test Group, Float Test Group, Bool Test Group, String Test Group, Date Test Group

4.1.3 Less Than (<)

The less than operation returns a Bool object. The returned Bool object will be 'T' if the first operand is less than the second operand, otherwise 'F' is returned.

The less than operation is implemented via the Utility Class as a static method.

Example Code:

```
print(1 < 2);     // T
print(1 < 0);     // F
print(1 < 1);     // F
print("1" < 1);   // F
print("1" < "2"); // T
print("1" < "0"); // F
```

Test Cases: Int Test Group, Float Test Group, Bool Test Group, String Test Group, Date Test Group

4.1.4 Greater Than (>)

The greater than operation returns a Bool object. The returned Bool object will be 'T' if the first operand is greater than the second operand, otherwise 'F' is returned.

The greater than operation is implemented via the Utility Class as a static method.

Example Code:

```
print(1 > 2);     // F
print(1 > 0);     // T
print(1 > 1);     // F
print("1" > 1);   // F
print("1" > "2"); // F
print("1" > "0"); // T
```

Test Cases: Int Test Group, Float Test Group, Bool Test Group, String Test Group, Date Test Group

4.1.5 Less Than or Equal To (<=)

The less than or equal to operation returns a Bool object. The returned Bool object will be 'T' if the first operand is less than the second operand or both operands are equal. Otherwise 'F' is returned.

The less than or equal to operation is implemented via the Utility Class as a static method.

Example Code:

```
print(1 <= 2);     // T
print(1 <= 0);     // F
print(1 <= 1);     // T
print("1" <= 1);   // T
print("1" <= "2"); // T
print("1" <= "0"); // F
```

Test Cases: Int Test Group, Float Test Group, Bool Test Group, String Test Group, Date Test Group

4.1.6 Greater Than or Equal To (>=)

The greater than or equal to operation returns a Bool object. The returned Bool object will be 'T' if the first operand is greater than the second operand or both operands are equal. Otherwise 'F' is returned.

The greater than or equal to operation is implemented via the Utility Class as a static method.

Example Code:

```
print(1 >= 2);     // F
print(1 >= 0);     // T
print(1 >= 1);     // T
print("1" >= 1);   // T
print("1" >= "2"); // F
print("1" >= "0"); // T
```

Test Cases: Int Test Group, Float Test Group, Bool Test Group, String Test Group, Date Test Group

4.2 Numeric Operations

Numeric operations are operations that apply to Numeric data types (see Section 2.1).

The behavior of numeric operations is dependent on the left operand. If the left operand is an Int, then the second operand (for binary operations) will be coerced into an Int if it is a Float. If the left operand is a Float, then the second operand (for binary operations) will be coerced into a Float if it is an Int.

4.2.1 Binary Plus (+)

The binary plus operation takes two numeric values (Int or Float data type) and returns the result of adding the second numeric to the first numeric. The right operand will be coerced to the data type of the left operand if necessary, and the result will have the same data type as the left operand.

The binary plus operation is implemented via the Utility Class as a static method.

Example Code:

```
Int iVal = 10;
Float fVal = 1.25;

print(iVal + 8);    // 18
print(iVal + 2.75); // 12
print(fVal + 3);    // 4.25
print(fVal + 0.25); // 1.5
```

Test Cases: Int Test Group, Float Test Group, Bool Test Group, String Test Group, Expression Test Group

4.2.2 Binary Minus (-)

The binary minus operation takes two numeric values (Int or Float data type) and returns the result of subtracting the second numeric from the first numeric. The right operand will be coerced to the data type of the left operand if necessary, and the result will have the same data type as the left operand.

The binary minus operation is implemented via the Utility Class as a static method.

Example Code:

```
Int iVal = 10;
Float fVal = 1.25;

print(iVal - 8);    // 2
print(iVal - 2.75); // 8
print(fVal - 3);    // -1.75
print(fVal - 0.25); // 1.0
```

Test Cases: Int Test Group, Float Test Group, Bool Test Group, String Test Group, Expression Test Group

4.2.3 Unary Minus (-)

The unary minus operation takes a single numeric value and returns the result of negating the value of the numeric operand. The result will have the same data type as the operand provided.

The unary minus operation is implemented via the Utility Class as a static method.

Example Code:

```
Int iVal = -25;
Float fVal = -0.25;

print(-iVal); // 25
print(-12);   // -12
print(-fVal); // 0.25
print(-0.33); // -0.33
```

Test Cases: Int Test Group, Float Test Group, Bool Test Group, String Test Group, Expression Test Group

4.2.4 Multiplication (*)

The multiplication operation takes two numeric values (Int or Float data type) and returns the result of multiplying the second numeric by the first numeric. The right operand will be coerced to the data type of the left operand if necessary, and the result will have the same data type as the left operand.

The multiplication operation is implemented via the Utility Class as a static method.

Example Code:

```
Int iVal = 25;
Float fVal = 0.25;

print(iVal * 5);    // 125
print(iVal * 3.9);  // 75
print(fVal * 2);    // 0.5
print(fVal * 0.25); // 0.0625
```

Test Cases: Int Test Group, Float Test Group, Bool Test Group, String Test Group, Expression Test Group

4.2.5 Division (/)

The division operation takes two numeric values (Int or Float data type) and returns the result of dividing the second numeric from the first numeric. The right operand will be coerced to the data type of the left operand if necessary, and the result will have the same data type as the left operand.

The division operation is implemented via the Utility Class as a static method.

Example Code:

```
Int iVal = 125;
Float fVal = 15.25;

print(iVal / 25);   // 5
print(iVal / 15.9); // 8
print(fVal / 2);    // 7.625
print(fVal / 0.25); // 61.0
```

Test Cases: Int Test Group, Float Test Group, Bool Test Group, String Test Group, Expression Test Group

4.2.6 Exponentiation (^)

The exponentiation operation takes two numeric values (Int or Float data type) and returns the result of raising the first numeric to the power of the second numeric. The right operand will be coerced to the data type of the left operand if necessary, and the result will have the same data type as the left operand.

The exponentiation operation is implemented via the Utility Class as a static method.

Example Code:

```
Int iVal = 10;
Float fVal = 0.25;

print(iVal ^ 2);    // 100
print(iVal ^ 0.1);  // 1
print(fVal ^ 2);    // 0.0625
print(fVal ^ 3.0);  // 0.015625
```

Test Cases:

4.2.7 Addition and Subtraction Assignment

The addition and subtraction assignment operations take one numeric expression and assigns the result of that operand added (+=) or subtracted (-=) from the original identifier's value.

Example Code:

```
Int iValue = 5;
iValue += 5;
print(iValue); //10
iValue -= 15;
print(iValue); //0
```

Test Cases: Int Test Group, Float Test Group, Bool Test Group, String Test Group, Expression Test Group

4.3 String Operations

4.3.1 Concatenation (#)

The String concatenation operator concatenates two Strings into a single String, where the second String is concatenated to the first String immediately after the last character in the first String.

Both String objects and constants can be concatenated.

The String concatenation operation is implemented via the Utility Class as a static method.

Example Code:

```
String str1 = "King";
print("Burger " # str1);  // Burger King
print("School " # "Bus"); // School Bus
```

Test Cases: String Test Group

4.3.2 Subscripting (*string*[i])

String subscripting provides a way to index individual characters within a string. String characters are indexed starting from 0, where index 0 is the first character. Our implementation of subscripting allows negative indexing of strings starting from the last character, where -1 is the last character in the string.

The string subscripting operation is implemented via the Utility Class as a static method.

Example Code:

```
String str = "abcdefg";
print(str[0]);  // a
print(str[1]);  // b
print(str[-1]); // g
print(str[-2]); // f
```

Test Cases: String Test Group

4.4 Array Operations

4.4.1 Subscripting (*array*[i])

Array subscripting provides a way to index individual elements within an array. Array elements are numbered starting from 0, where index 0 is the first element in the array. Our implementation of subscripting allows negative indexing of arrays starting from the last element, where -1 is the last element in the array.

The array subscripting operation is implemented via the Utility Class as a static method.

Example Code:

```
Int array[] = 10, 20, 30, 40, 50, 60;
print(array[0]);  // 10
print(array[1]);  // 20
print(array[-1]); // 60
print(array[-2]); // 50
```

Test Cases: All Test Groups

4.5 Logical Operations

4.5.1 Logical And (and)

The logical and operation requires two Bool operands and returns a Bool object. The returned Bool object's value is 'T' if and only if both operands are 'T'. Otherwise 'F' is returned.

The logical and operation is implemented via the Utility Class as a static method.

Example Code:

```
print(T and T); // T
print(T and F); // F
print(F and T); // F
print(F and F); // F
```

Test Cases: Bool Test Group, Expression Test Group

4.5.2 Logical Or (or)

The logical or operation requires two Bool operands and returns a Bool object. The returned Bool object's value is 'T' either or both operands are 'T'. Otherwise 'F' is returned.

The logical or operation is implemented via the Utility Class as a static method.

Example Code:

```
print(T or T); // T
print(T or F); // T
print(F or T); // T
print(F or F); // F
```

Test Cases: Bool Test Group, Expression Test Group

4.5.3 Logical Not (not)

The logical or operation requires a Bool operand and returns a Bool object. The returned Bool object's value is 'T' if the Bool operand is 'F', or 'F' if the Bool operand is 'T'.

The logical not operation is implemented via the Utility Class as a static method.

Example Code:

```
print(not F); // T
print(not T); // F
```

Test Cases: Bool Test Group, Expression Test Group

5 Slices

Our implementation of pickle supports the slicing of Strings and Arrays. Slicing is the process of creating substrings or sub-arrays of Strings or Arrays.

5.1 Slicing Strings

Strings can be sliced by combining the subscript operation '[]' and the slicing operation '~'. A String is sliced from an inclusive lower-bound index to an exclusive upper-bound index. The indices provided must be within the bounds of the String (upper-bound can be max+1 since it is exclusive). Index values start at 0, which represents the index of the first character in the String.

It is not necessary to provide both the lower-bound and upper-bound indices if you are slicing from the beginning or end of a String. Either the lower-bound or the upper-bound index can be blank, where a blank lower-bound represents the index 0 and a blank upper-bound represents the index of the last character.

Strings can be assigned into slices of Strings, which will grow or shrink the length of the target String based on the size of the slice and the length of the source String. Assignment uses the following rules:

- When the source String is larger than the length of the target slice, the target String will grow to fit the entire source String within the slice.
- When the source String is smaller than the length of the target slice, the target String will shrink.

Example Code:

```
String str;

str = "goodbye";
print(str[0~4]); // good
print(str[~4]);  // good
print(str[4~]);  // bye

str = "tacobell";
str[~4] = "school";
print(str);         // schoolbell

str[0~6] = "";
print(str);         // bell
```

Test Cases: String Test Group, Expression Test Group

5.2 Slicing Arrays

Arrays can be sliced by combining the subscript operation '[]' and the slicing operation '~'. An array is sliced from an inclusive lower-bound index to an exclusive upper-bound index. The indices provided must be within the bounds of the array (upper-bound can be max+1 since it is exclusive). Index values start at 0, which is the index of the first element in the array.

It is not necessary to provide both the lower-bound and upper-bound indices if you are slicing from the beginning or end of an array. Either the lower-bound or the upper-bound index can be blank, where a blank lower-bound represents the index 0 and a blank upper-bound represents the index of the last element.

Arrays cannot be assigned into slices of Arrays.

Example Code:

```
Float gradeM [] = 90.5, 50.0, 60.0, 85.5;
Float myGradeM[5];

print(gradeM[2~3]); // 60.0
print(gradeM[~3]);  // 90.5 50.0 60.0
print(gradeM[2~]);  // 60.0 85.5

// The following assignment would cause
// myGradeM to contain 60.0 and 85.5
myGradeM = gradeM[2~];

print(myGradeM); // 60.0 85.5
```

Test Cases: Int Test Group, Float Test Group, Bool Test Group, String Test Group, Expression Test Group

6 Flow Control

6.1 break and continue statements:

Users can use the break and continue statements to alter the flow control of for or while loops. Using break to exit the loop and using continue to skip to the next iteration without running anymore code.

Example Code:

```
Int arr[] = 1, 2, 3, 4, 5;
for i in arr:                       // prints:
       if i == 3:                           //          1
              continue;             //          2
       endif;                       //          4
       print(i);                            //          5
endfor;
```

Test Cases: Flow Test Group

6.2 for *cv* = *sv* to *endValue* by *incr*:

Users can use a for statement to loop for a set number of instances by a certain increment if they choose so. An optional by keyword can be used to set the increment value. If the by keyword is not set an increment of 1 will be used. The loop must be ended with endfor.

Example Code:

```
for i = 0 to 5 by 2:        //prints:
       print(i);            //          0
endfor;             //          2
                    //          4
for i = 1 to 5:             //prints:
       print(i);            //          1
endfor;             //          2
                    //          3
                    //          4
```

Test Cases:

6.3 for *char* in *string*:

Users can use a for statement to loop through all characters in a String. Slices can be used. The loop must be ended with endfor.

Example Code:

```
for c in “Howdy”:            // prints “Howdy” with each character being on a new line
       print(c);
endfor;
String str = “Hello”;
for c in str:         // prints “Hello” with each character being on a new line
       print(c);
endfor;
for c in str[~2]:            // prints “He” with each character being on a new line
       print(c);
endfor;
```

Test Cases: Flow Test Group

6.4 for *item* in *array*:

Users can use a for statement to loop through each element in an array. Slices can be used. The loop must be ended with endfor.

Example Code:

```
Bool arr[] = T, F, F, T, F;
for b in str:         // prints “TFFTF” with each character being on a new line
       print(b);
endfor;
for b in arr[~2]:            // prints “TF” with each character being on a new line
       print(b);
endfor;
```

Test Cases: Flow Test Group

6.5 for *stringCV* from *string* by *delimiter*:

Users can use a for statement to effectively tokenize a string. If no delimiter is found, the control variable will be the entire *string* and the loop will end after one iteration. The loop must be ended with endfor.

Example Code:

```
for token from “apple, clark, hello” by “,”:            //prints:
       print(token);                        //           apple
endfor;                                     //           clark
                                            //           hello
```

Test Cases: Flow Test Group

6.6 while *condition*:

Users can use a while statement to repeat statements as long as a certain condition evaluates to true. The loop must be ended with endwhile.

Example Code:

```
Bool b = T;
Int i = 1;
while b:                                //prints:
       if i == 4:                       //            1
              b = F;            //            2
       endif;                   //            3
       print(i);                        //            4
       i += 1;
endwhile;
```

Test Cases: Flow Test Group

6.7 if *condition*:

Users can use an if statement to execute a certain block of code if a certain condition evaluates to true. The block must end with endif. (See 6.8 for an exception)

Example Code:

```
Bool b = T;
if b == T:
       print(“True”);        //prints True
endif;
if b == F:
       print(“False”);              // doesn’t enter code block and therefore does not
print
endif;
```

Test Cases: Flow Test Group

6.8 else

An else statement can be entered after an if statement and before the endif statement in order to execute a block of code only if the condition in the if statement is false.

Example Code:

```
Bool b = F;
if b == T:
       print(“True”);        // doesn’t enter code block and therefore doesn’t print
else:
       print(“False”);              // the condition of the if statement evaluated to
false, so
endif;                       // the block of code below the else is executed, printing
“False”
```

Test Cases: Flow Test Group

6.9 select when default

The user can use the select statement to check a value against a list of different values of the same type and execute statements accordingly. To check values, the user uses a when statement with the value(s) to check. An optional default statement can be used to execute code if none of the previous when values were evaluated as true.

Example Code:

```
Int i = 4;
select i:                                   //prints 4
       when 0, 1:
              print(“0 or 1”);
       when 2:
              print(“2”);
       when 3:
              print(“3”);
       when 4:
              print(“4”);
       when 5, 6, 7, 8, 9:
              print(“5, 6, 7, 8, 9”);
       default:
              print(“Not 0-9”);
endselect;
i = 10;
select i:                                   //prints NOT 0-9
       when 0, 1:
              print(“0 or 1”);
       when 2:
              print(“2”);
       when 3:
              print(“3”);
       when 4:
              print(“4”);
       when 5, 6, 7, 8, 9:
              print(“5, 6, 7, 8, 9”);
       default:
              print(“Not 0-9”);
endselect;
```

Test Cases: Flow Test Group

7 Functions

7.1 Built-In Functions

7.1.1 print()

The built-in function print() prints output to standard out. A variable number of arguments can be listed in the print() function, and each item will be printed separated by a space. The function automatically includes a newline after printing all provided arguments. When printing arrays, only populated elements will be printed with a single space separating each element.

Printing of each data type uses specific coercion rules as defined in Section 2.

Example Code:

```
String str = “is printed”;
Int iVal = 24;
Float fVal = 0.25;
Bool bFlag = T;
Date today = “2021-05-01”;
Int array[3] = 1, 2, 3;

print(str);    // is printed
print(“Test”); // Test

print(iVal);   // 24
print(10);     // 10

print(fVal);   // 0.25
print(3.14);   // 3.14

print(bFlag);  // T
print(F);      // F

print(today);  // 2021-05-01

print(“This”, str, “!”);       // This is printed!
print(“Array values:”, array); // Array Values: 1 2 3
```

Test Cases: Flow Test Group

7.1.2 LENGTH(*string*)

The built-in function LENGTH() returns the number of characters in the provided String. LENGTH() returns an Int value. It takes a single argument of type String.

This function can be used with both String objects and Sting constants.

Example Code:

```
String str = “Bubby”;

print(LENGTH(str)); // 5
```

Test Cases: String Test Group

7.1.3 SPACES(*string*)

The built-in function SPACES() returns a Bool value of 'T' if the provided String is empty or only contains spaces, or 'F' otherwise. It takes a single argument of type String.

This function can be used with both String objects and Sting constants.

Example Code:

```
String str = "not empty";

print(SPACES(str));   // F
print(SPACES(""));    // T
print(SPACES("   ")); // T
```

Test Cases: String Test Group

7.1.4 ELEM(*array*)

The built-in function ELEM() returns the subscript of the highest allocated element of an array +1 as an Int value. It takes a single argument of type array.

This function can be used with both bounded and unbounded arrays.

Example Code:

```
Int array[8] = 10, 20, 30, 40, 50, 60;
Int unboundedArray[unbound];
unboundedArray[15] = 10;

print(ELEM(array));    // 6

print(ELEM(unboundedArray));    // 15
```

Test Cases: All Test Groups

7.1.5 MAXELEM(array)

The built-in function MAXELEM() returns the declared number of elements of an array as an Int value. It takes a single argument of type array.

This function can be used with both bounded and unbounded arrays, but unbounded arrays will always return the same value as ELEM() because their capacity is dependent on the highest indexed allocated element.

Example Code:

```
Int array[8] = 10, 20, 30, 40, 50, 60;
Int unboundedArray[unbound];
unboundedArray[15] = 10;

print(ELEM(array));    // 6
print(MAXELEM(array)); // 8

print(ELEM(unboundedArray));    // 15
print(MAXELEM(unboundedArray)); // 15
```

Test Cases: All Test Groups

7.1.6 dateDiff(*date1*, *date2*)

The built-in function dateDiff() returns the difference between two dates in days as an Int value. It takes two arguments of type Date. The second date is subtracted from the first date to calculate the result.

The function can take a Date object or a String constant, if a String constant is provided it must be coercible into a Date object.

Example Code:

```
Date date = "1999-02-27";

print(dateDiff("2021-02-27", date), "days"); // 8036 days
```

Test Cases: Date Test Group

7.1.7 dateAdj(*date*, *int*)

The built-in function dateAdj() returns a new Date value adjusted by a number of days. It takes two arguments, the first of type Date and the second of type Int. The Int value provided can be positive or negative, where a positive value will be added to the Date and a negative value will be subtracted.

The function can take a Date object or a String constant, if a String constant is provided it must be coercible into a Date object.

Example Code:

```
Date date = "1999-02-27";

print(dateAdj(date, 10));                    // 1999-03-09d
```

Test Cases: Date Test Group

7.1.8 dateAge(*date1*, *date2*)

The built-in function dateAge() returns the difference between two dates in years as an Int Value. It takes two arguments of type Date. The second date is subtracted from the first date to calculate the result. The result will be the difference in years, rounded down to the full year if there are extra days.

The function can take a Date object or a String constant, if a String constant is provided it must be coercible into a Date object.

Example Code:

```
Date date = "1999-02-27";

print(dateAge("2021-02-27", date), "years"); // 22 years
```

Test Cases: Date Test Group

7.1.9 IN and NOTIN functions

Users can check to see if a value is in a list or not by utilizing the IN and NOTIN functions. IN returns T (true) if the value is in the list and F (false) if it is not in the list. Likewise NOTIN returns F (false) if the value is in the list and T (true) if the value is in the list.

The IN and NOTIN functions can only be used in place of *conditions*. Using IN or NOT in in an expression (like print) is not supported.

Example Code:

```
String fruit = “apple”;
if fruit IN {“apple”, “orange”, “pear”}: //prints “IN”
       print(“IN”);
else:
       print(“NOT IN”);
endif;
Int x = 5;
Int arr[] = 1, 2, 3, 4;
if x NOTIN arr:              //prints “NOT IN”
       print(“NOT IN”);
endif;
```

Test Cases: All Test Groups

7.2 User Defined Functions

7.2.1 Scoping

User defined functions utilize static scoping when assigning variable values.

Example Code:

```
def Void A():
	Int x;
	Int y;
	Int z;

	def Void D():
		Int w;
		print(“D before assign”, “x=”, x, “y=”, y, “z=”, z); //D before assign x=
30 y= 11 z= 12
		x = 40;
		y = 41;
		w = 44;
		print(“D after assign”, “x=”, x, “y=”, y, “z=”, z); //D after assign x= 40
y= 41 z= 12
	enddef; //end of D

	def Void B():
		Int z;
		Int y;

		def Void C():
			Int y;
			x = 30;
			y = 31;
			z = 32;
			print(“C before”, “x=”, x, “y=”, y, “z=”, z); //C before x= 30 y=
31 z= 32
			D();
			print(“C after”, “x=”, x, “y=”, y, “z=”, z); //C after x= 40 y= 31
z=32
		enddef; //end of C

		x = 20;
		y = 21;
		z = 22;
		print(“B before”, “x=”, x, “y=”, y, “z=”, z); //B before x= 20 y= 21 z= 22
		C();
		print(“B after”, “x=”, x, “y=”, y, “z=”, z); //B after x= 40 y= 21 z= 32
	enddef; //end of B

	x = 10;
	y = 11;
	z = 12;
	print(“A before”, “x=”, x, “y=”, y, “z=”, z); //A before x= 10 y= 11 z= 12
	B();
	print(“A after”, “x=”, x, “y=”, y, “z=”, z); //A after x= 40 y= 41 z= 12
	enddef; //end of A
A();
```

Test Cases: Function Test Group

7.2.2 Return and Parameter Types

User defined functions can return types of Void, Int, Bool, Float, Date as well as Strings and arrays (including slices). User defined function parameters can be primitive types of Int, Bool, Float, Date as well as Strings and arrays (including slices).

7.2.2.1 Void (Return type only)

The Void type is used as a return type only and signifies that no value will be return by the function.

Example Code:

```
def Void printInt(Int x):
       print("Int value is: ", x); // Int value is: 17
       return; //not necessary to include a return statement when return type is Void
enddef;

printInt(17);
```

Test Cases: Function Test Group

7.2.2.2 Int

The Int type is used as both a return type and a parameter type. As a return type it is used to signify that an integer will be returned to the caller by the function. As a parameter it is used to signify that an integer will be passed to the function by the callee.

Example Code:

```
def Int printIntPlusTwo(Int x):
       print("Int value is: ", x); // Int value is: 17
       return x + 2;
enddef;

Int y = printIntPlusTwo(17);
print(y); //19
```

Test Cases: Function Test Group

7.2.2.3 Bool

The Bool type is used as both a return type and a parameter type. As a return type it is used to signify that a boolean will be returned to the caller by the function. As a parameter it is used to signify that a boolean will be passed to the function by the callee.

Example Code:

```
def Bool isTrue(Bool b):
	if b:
		return T;
	else:
		return F;
	endif
enddef;

print(isTrue(T)); // T
print(isTrue(F)); // F
```

Test Cases: Function Test Group

7.2.2.4 Float

The Float type is used as both a return type and a parameter type. As a return type it is used to signify that a float will be returned to the caller by the function. As a parameter it is used to signify that a float will be passed to the function by the callee.

Example Code:

```
def Float addFloats(Float x, Float y):
	return x + y;
enddef;

Float x = addFloats(1.5, 2.0);
print("value: ", x); // value: 3.5
```

Test Cases: Function Test Group

7.2.2.5 Date

The Date type is used as both a return type and a parameter type. As a return type it is used to signify that a date will be returned to the caller by the function. As a parameter it is used to signify that a date will be passed to the function by the callee.

Example Code:

```
def Date addDays(Date date, Int x):
	return dateAdj(date, x);
enddef;

Date d = addDays("1998-06-21", 15);
print("value: ", d); // 1998-07-06
```

Test Cases: Function Test Group

7.2.2.6 Strings / Slices

The String type is used as both a return type and a parameter type. As a return type it is used to signify that a string will be returned to the caller by the function. As a parameter it is used to signify that a string will be passed to the function by the callee. Either the return type String or the parameter String can be sliced to return/pass a modified String.

Example Code:

```
def String swapString(String str):
       str = “new String”;
       return str;
enddef;

def String sliceString(String str, Int x):
       return str[~x];
enddef;

String string = “orig String”;
String string2 = swapString(string);

print(string, “:”, string2); // new String : new String
print(sliceString(“Hi clark”, 3));
```

Test Cases: Function Test Group

7.2.2.7 Arrays / Slices

An array can be used as both a return type and a parameter type. As a return type it is used to signify that an array will be returned to the caller by the function. As a parameter it is used to signify that an array will be passed to the function by the callee. Either the return type array or the parameter array can be sliced to return/pass a modified array.

Example Code:

```
def Float[] createArr(Float x):
	Float arr[5];
	arr = x;
	return arr[2~];
enddef;

def Int[] combineSlices(Int x[], Int y[]):
	Int arr[MAXELEM(x) + MAXELEM(y)];
	Int i = 0;
	for Int j in x:
		arr[i] = x[j];
		i += 1;
	endfor;
	for Int j in y:
		arr[i] = y[j];
		i += 1;
	endfor;
	return arr;
enddef;

print(createArr(12.2)); // 12.2 12.2 12.2
Int arr1[] = 1, 2, 3, 4, 5;
Int arr2[] = 6, 7, 8, 9, 10;
print(combineSlices(arr1[2~], arr2[~3])); // 3, 4, 5, 6, 7, 8
```

Test Cases: Function Test Group

7.2.3 Recursion

Recursion is possible with user defined functions.

Example Code:

```
def Int factorial(Int x):
	if x == 0:
		return 1;
	else:
		return x * factorial(x – 1);
	endif;
enddef;

print(“value: “, factorial(10)); // value: 3628800
```

Test Cases: Function Test Group

7.2.4 Parameters by value and by reference

By default, all parameters in user defined functions are pass by reference. The user can specify by value or by reference explicitly if desired. If using by value, the value of the variable passed to the function will not be changed in the outer scope, only inside the function. If using by reference, the value of the variable will be changed in the outer scope.

Example Code:

```
def Void func(Int x, Value Int y, Ref Int z, Value Int arr[], int expr):
       x = x + 5;
       y = y + 5;
       z = z + 5;
       arr[1] = 20;
       expr = -1;
enddef;

Int a = 5;
Int b = 5;
Int c = 5;
Int array[] = 5, 10;
func(a, b, c, array, 13 + 6);
print(“a=”, a, “b=”, b, “c=”, c, “array=”, array); //a= 10 b= 5 c= 10 array= 5 10
```

Test Cases: Function Test Group

8 Flexible Points Completed

Flexible Feature	Points	Completed (yes, -)	Information
Date	**10**	**yes**	**required of everyone**
break and continue	**10**	**yes**	**required of everyone**
Infix Expressions	25	yes	
Unbounded Arrays	5	yes	
Additional Numeric Assignment	5	yes	+=, -=
IN, NOTIN	10	yes	
slices	30	yes	
+5 if doing infix and slices	5	yes	test cases must show many uses of this
for tokenizing	10	yes	
select when default	10	yes	including break and continue
programmer-defined functions	80	yes	
functions by value	5	yes	in addition to by reference
functions variable number of args	10	-	
functions return arrays	5	yes	if slices were done, also returning slices
Excellent test Cases	**30**	yes	**You must create test cases which completely test your interpreter. This includes testing for error cases. If your cases are inadequate, you can lose an additional 150 points.**
Total Completed (out of 250)	240	yes	

9 Test Files

Test files are organized into groups: Int, Float, Bool, String, Date, Expression, Flow and Function Tests.

The sections below list each test file organized by group, the name of the file serves as the description of the file contents.

9.1 Int Tests

1 - Int_declaration.txt
2 - Int_assignment.txt
3 - Int_const_equal_to_Int_const_operation.txt
4 - Int_var_equal_to_Int_const_operation.txt
5 - Int_var_equal_to_Int_var_operation.txt
6 - Int_var_not_equal_to_Int_const_operation.txt
7 - Int_var_not_equal_to_Int_var_operation.txt
8 - Int_const_not_equal_to_Int_const_operation.txt
9 - Int_var_less_than_Int_const_operation.txt
10 - Int_var_less_than_Int_var_operation.txt
11 - Int_const_less_than_Int_const_opertaion.txt
12 - Int_var_greater_than_Int_const_operation.txt
13 - Int_var_greater_than_Int_var_operation.txt
14 - Int_const_greater_than_Int_const_operation.txt
15 - Int_var_less_than_or_equal_Int_const_operation.txt
16 - Int_var_less_than_or_equal_Int_var_operation.txt
17 - Int_const_less_than_or_equal_Int_const_operation.txt
18 - Int_var_greater_than_or_equal_Int_const_operation.txt
19 - Int_var_greater_than_or_equal_Int_var_operation.txt
20 - Int_const_greater_than_or_equal_Int_const_operation.txt
21 - Int_const_binary_plus_Int_const_operation.txt
22 - Int_var_binary_plus_Int_const_operation.txt
23 - Int_var_binary_plus_Int_var_operation.txt
24 - Int_const_binary_minus_Int_const_operation.txt
25 - Int_var_binary_minus_int_const_operation.txt
26 - Int_var_binary_minus_int_var_operation.txt
27 - Int_const_unary_minus_operation.txt

28 - Int_var_unary_minus_operation.txt
29 - Int_const_multiplication_Int_const_operation.txt
30 - Int_var_multiplication_Int_const_operation.txt
31 - Int_var_multiplication_Int_var_operation.txt
32 - Int_const_division_Int_const_operation.txt
33 - Int_var_division_Int_const_operation.txt
34 - Int_var_division_Int_var_operation.txt
35 - Int_const_exponentiation_Int_const_operation.txt
36 - Int_var_exponentiation_Int_const_operation.txt
37 - Int_var_exponentiation_Int_var_operation.txt
38 - Int_const_addition_assignment_operation.txt
39 - Int_var_addition_assignment_operation.txt
40 - Int_const_subtraction_assignment_operation.txt
41 - Int_var_subtraction_assignment_operation.txt
42 - Int_coerce_Float.txt
43 - Int_coerce_String.txt
44 - Int_for_cv_to_endVal.txt
45 - Int_for_item_in_arr.txt
46 - Int_parameters.txt
47 - Int_returning.txt
48 - Int_slices.txt
49 - Int_IN.txt
50 - Int_NOTIN.txt
51 - Int_print.txt
52 - Int_select.txt
53 - Int_logical_and.txt
54 - Int_logical_or.txt
55 - Int_logical_not.txt
56 - Int_While.txt

9.2 Float Tests

1 - Float_Declaration.txt
2 - Float_Assignment_from_Numeric_Constant_Float.txt
3 - Float_Assignment_from_Numeric_Constant_Int.txt

4 - Float_Assignment_from_String.txt
5 - Float_var_Equal_To_Float_var_operation.txt
6 - Float_const_Equal_To_Float_var_operation.txt
7 - Float_var_Equal_To_Float_const_operation.txt
8 - Float_const_Equal_To_Float_const_operation.txt
9 - Float_var_Not_Equal_To_Float_var_operation.txt
10 - Float_const_Not_Equal_To_Float_var_operation.txt
11 - Float_var_Not_Equal_To_Float_const_operation.txt
12 - Float_const_Not_Equal_To_Float_const_operation.txt
13 - Float_var_Less_Than_Float_var_operation.txt
14 - Float_const_Less_Than_Float_var_operation.txt
15 - Float_var_Less_Than_Float_const_operation.txt
16 - Float_const_Less_Than_Float_const_operation.txt
17 - Float_var_Greater_Than_Float_var_operation.txt
18 - Float_const_Greater_Than_Float_var_operation.txt
19 - Float_var_Greater_Than_Float_const_operation.txt
20 - Float_const_Greater_Than_Float_const_operation.txt
21 - Float_var_Less_Than_Or_Equal_To_Float_var_operation.txt
22 - Float_const_Less_Than_Or_Equal_To_Float_var_operation.txt
23 - Float_var_Less_Than_Or_Equal_To_Float_const_operation.txt
24 - Float_const_Less_Than_Or_Equal_To_Float_const_operation.txt
25 - Float_var_Greater_Than_Or_Equal_To_Float_var_operation.txt
26 - Float_const_Greater_Than_Or_Equal_To_Float_var_operation.txt
27 - Float_var_Greater_Than_Or_Equal_To_Float_const_operation.txt
28 - Float_const_Greater_Than_Or_Equal_To_Float_const_operation.txt
29 - Float_var_Binary_Plus_Float_var_operation.txt
30 - Float_const_Binary_Plus_Float_var_operation.txt
31 - Float_var_Binary_Plus_Float_const_operation.txt
32 - Float_const_Binary_Plus_Float_const_operation.txt
33 - Float_var_Binary_Minus_Float_var_operation.txt
34 - Float_const_Binary_Minus_Float_var_operation.txt
35 - Float_var_Binary_Minus_Float_const_operation.txt
36 - Float_const_Binary_Minus_Float_const_operation.txt

37 - Float_var_Unary_Minus_Assignment_operation.txt
38 - Float_var_Unary_Minus_Float_var_operation.txt
39 - Float_var_Unary_Minus_Float_const_operation.txt
40 - Float_var_Multiplication_Float_var_operation.txt
41 - Float_const_Multiplication_Float_var_operation.txt
42 - Float_var_Multiplication_Float_const_operation.txt
43 - Float_const_Multiplication_Float_const_operation.txt
44 - Float_var_Division_Float_var_operation.txt
45 - Float_const_Division_Float_var_operation.txt
46 - Float_var_Division_Float_const_operation.txt
47 - Float_const_Division_Float_const_operation.txt
48 - Float_var_Exponentiation_Float_var_operation.txt
49 - Float_const_Exponentiation_Float_var_operation.txt
50 - Float_var_Exponentiation_Float_const_operation.txt
51 - Float_const_Exponentiation_Float_const_operation.txt
52 - Float_const_Addition_Assignment_operation.txt
53 - Float_var_Addition_Assignment_operation.txt
54 - Float_const_Addition_Assignment_operation.txt
55 - Float_var_Addition_Assignment_operation.txt
56 - Float_for_cv_to_endVal.txt
57 - Float_for_item_in_array.txt
58 - Float_parameters.txt
59 - Float_returning.txt
60 - Float_slices.txt
61 - Float_IN.txt
62 - Float_NOTIN.txt
63 - Float_print.txt
64 - Float_select.txt
65 - Float_logical_and.txt
66 - Float_logical_or.txt
67 - Float_logical_not.txt
68 - Float_while.txt

9.3 Bool Tests

1 - Bool_declaration.txt
2 - Bool_assignment.txt
3 - Bool_const_equal_to_Bool_const_operation.txt
4 - Bool_var_equal_to_Bool_const_operation.txt
5 Bool_var_equal_to_Bool_var_operation.txt
6 - Bool_const_not_equal_to_Bool_const_operation.txt
7 - Bool_var_not_equal_to_Bool_const_operation.txt
8 - Bool_var_not_equal_to_Bool_var_operation.txt
9 - Bool_var_less_than_Bool_const_operation.txt
10 - Bool_var_less_than_Bool_var_operation.txt
11 - Bool_const_less_than_Bool_const_operation.txt
12 - Bool_var_greater_than_Bool_const_operation.txt
13 - Bool_var_greater_than_Bool_var_operation.txt
14 - Bool_const_greater_than_Bool_const_operation.txt
15 - Bool_var_less_than_or_equal_Bool_const_operation.txt
16 - Bool_var_less_than_or_equal_Bool_var_operation.txt
17 - Bool_const_less_than_or_equal_Bool_const_operation.txt
18 - Bool_var_greater_than_or_equal_Bool_const_operation.txt
19 - Bool_var_greater_than_or_equal_Bool_var_operation.txt
20 - Bool_const_greater_than_or_equal_Bool_const_operation.txt
21 - Bool_array_subscripting.txt
22 - Bool_const_logical_and_bool_const_operations.txt
23 - Bool_var_logical_and_bool_const_operations.txt
24 - Bool_var_logical_and_bool_var_operations.txt
25 - Bool_var_logical_or_bool_const_operations.txt
26 - Bool_var_logical_or_bool_var_operations.txt
27 - Bool_const_logical_and_bool_const_operations.txt
28 - Bool_var_logical_not_operations.txt
29 - Bool_const_logical_not_operations.txt
30 - Bool_array_slicing.txt
31 - Bool_for_item_array.txt
32 - Bool_while_loop.txt

33 - Bool_select.txt
34 - Bool_IN.txt
35 - Bool_NOTIN.txt
36 - Bool_parameters.txt
37 - Bool_returning.txt
38 - Bool_coerce_String.txt
39 - Bool_assignment_bool_expr.txt

9.4 String Tests

1 - String_Declaration.txt
2 - String_Initialization_from_String_Constant_Double_Quotes.txt
3 - String_Initialization_from_String_Constant_Single_Quotes.txt
4 - String_Initialization_from_Numeric_Constant_Int.txt
5 - String_Initialization_from_Numeric_Constant_Int_Var.txt
6 - String_Initialization_from_Numeric_Constant_Float.txt
7 - String_Initialization_from_Numeric_Constant_Float_Var.txt
8 - String_Initialization_from_Date_Var.txt
9 - String_Initialization_from_Bool_Var.txt
10 - String_Initialization_from_StringConst_StringConst_Concat.txt
11 - String_Initialization_from_StringConst_StringVar_Concat.txt
12 - String_Initialization_from_StringVar_StringVar_Concat.txt
13 - String_Initialization_from_StringConst_IntConst_Concat.txt
14 - String_Initialization_from_StringConst_IntVar_Concat.txt
15 - String_Initialization_from_StringVar_IntConst_Concat.txt
16 - String_Initialization_from_StringVar_IntVar_Concat.txt
17 - String_Initialization_from_String_Slice_Lower_Bound_Only.txt
18 - String_Initialization_from_String_Slice_Upper_Bound_Only.txt
19 - String_Initialization_from_String_Slice_Lower_Upper_Bound.txt
20 - String_Slice_Assignment_from_Lower_Bound_Only.txt
21 - String_Slice_Assignment__from_Upper_Bound_Only.txt
22 - String_Slice_Assignment_from_Lower_Upper_Bound.txt
23 - String_Array_Initialization_from_String_Constant.txt
24 - String_Array_Initialization_from_Numeric_Constant_Int.txt
25 - String_Array_Initialization_from_Numeric_Constant_Float.txt

26 - StringVar_Equal_To_StringVar.txt
27 - StringVar_Equal_To_StringConst.txt
28 - StringConst_Equal_To_StringVar.txt
29 - StringConst_Equal_To_StringConst.txt
30 - StringVar_NotEqual_To_StringConst.txt
31 - StringConst_NotEqual_To_StringVar.txt
32 - StringConst_NotEqual_To_StringConst.txt
33 - String_Constant_IN_String_Array.txt
34 - String_Var_IN_String_Array.txt
35 - String_Constant_IN_String_List.txt
36 - String_Var_IN_String_List.txt
37 - String_Constant_NOTIN_String_Array.txt
38 - String_Var_NOTIN_String_Array.txt
39 - String_Constant_NOTIN_String_List.txt
40 - String_Var_NOTIN_String_List.txt

9.5 Date Tests

1 - Date_declaration.txt
2 - Date_assignment.txt
3 - Date_var_equal_to_Date_const_operation.txt
4 - Date_var_equal_to_Date_var_operation.txt
5 - Date_const_not_equal_to_Date_const_operation.txt
6 - Date_var_not_equal_to_Date_const_operation.txt
7 - Date_var_not_equal_to_Date_var_operation.txt
8 - Date_const_not_equal_to_Date_const_operation.txt
9 - Date_var_less_than_Date_const_operation.txt
10 - Date_var_less_than_Date_var_operation.txt
11 - Date_const_less_than_Date_const_operation.txt
12 - Date_var_greater_than_Date_const_operation.txt
13 - Date_var_greater_than_Date_var_operation.txt
14 - Date_const_greater_than_Date_const_operation.txt
15 - Date_var_less_than_or_equal_Date_const_operation.txt

16 - Date_var_less_than_or_equal_Date_var_operation.txt
17 - Date_const_less_than_or_equal_Date_const_operation.txt
18 - Date_var_greater_than_or_equal_Date_const_operation.txt
19 - Date_var_greater_than_or_equal_Date_var_operation.txt
20 - Date_const_greater_than_or_equal_Date_const_operation.txt
21 - Date_array_subscripting.txt
22 - Date_logical_and.txt
23 - Date_logical_or.txt
24 - Date_logical_not.txt
25 - Date_select.txt
26 - Date_for_item_array.txt
27 - Date_while.txt
28 - Date_date_diif.txt
29 - Date_date_adj.txt
30 - Date_date_age.txt
31 - Date_IN.txt
31 - Date_NOTIN.txt
32 - Date_parameter.txt
33 - Date_returning.txt

9.6 Expression Tests

1 - Expression_simple.txt
2 - Expression_simple_with_fuction.txt
3 - Expression_complex.txt
4 - Expression_complex_with_function.txt
5 - Expression_nested_function_calls.txt
6 - Expression_simple_boolean.txt
7 - Expression_complex_boolean.txt

9.7 Flow Tests

1 - Flow_if.txt
2 - Flow_while.txt
3 - Flow_for_char_in_string.txt
4 - Flow_for_item_in_array.txt

5 - Flow_for_string_delimiter.txt
6 - Flow_for_sv_to_ev.txt
7 - Flow_select.txt
8 - Flow_break.txt
9 - Flow_continue.txt

9.8 Function Tests

1 - function_print.txt
2 - function_LENGTH.txt
3 - function_SPACES.txt
4 - function_ELEM.txt
5 - function_MAXELEM.txt
6 - function_dateDiff.txt
7 - function_dateAdj.txt
8 - function_dateAge.txt
9 - function_IN.txt
10 - function_NOTIN.txt
11 - function_user_scoping.txt
12 - function_user_return.txt
13 - function_user_parameters.txt
14 - function_user_recursion.txt
15 - function_user_by_reference_param.txt
15 - function_user_by_value_param.txt