

```

// [THIS IS READ-ONLY]
@file:DependsOn("/antlr-4.11.1-complete.jar")
@file:DependsOn("./target")

// [THIS IS READ-ONLY]
import org.antlr.v4.runtime.*
import backend.*

// [THIS IS READ-ONLY]
fun execute(source:String) {
    val errorlistener = object: BaseErrorListener() {
        override fun syntaxError(recognizer: Recognizer<*,*>,
            offendingSymbol: Any?,
            line: Int,
            pos: Int,
            msg: String,
            e: RecognitionException?) {
            throw Exception("${e} at line:${line}, char:${pos}")
        }
    }
    val input = CharStreams.fromString(source)
    val lexer = PLLexer(input).apply {
        removeErrorListeners()
        addErrorListener(errorlistener)
    }
    val tokens = CommonTokenStream(lexer)
    val parser = PLParser(tokens).apply {
        removeErrorListeners()
        addErrorListener(errorlistener)
    }

    try {
        val result = parser.program()
        result.expr.eval(Runtime())
    } catch (e:Exception) {
        println("Error: ${e}")
    }
}

```

Data Types Supported / Declarable Values

Integer type String type Boolean type List type (Integer and String) Functions type

Variable Declaration

Done through defining the (1) **Variable_Name**, (2) **Variable_Value** (3) and spacing it out with " = "
 " Lists can be declared the same way by using square bracket

Variable_name = **Variable_Value**

Note: Spacing for the some of the functions are sensitive

Example:

```
val program = """
print("String Type Value");
x = "Hello";
print(x);

print("Integer Type Value");
x = 20;
print(x);

print("String List Type Value");
x = ["Hello", "World", "."];
print(x);

print("Integer List Type Value");
x = [1, 100, 200, 30];
print(x);
"""
```

execute(program)

```
String Type Value
Hello
Integer Type Value
20
String List Type Value
[Hello, World, .]
Integer List Type Value
[1, 100, 200, 30]
```

Lists

Like other programming languages specific values stored inside of a list can be accessed. Where the index number goes from 0 -> n (number of items in the list)

Editing an Existing List

You can update an already declared list

array_name[**index_number**] = new_value

```
val program = """
x = [1, 2, 3, 4, 5];

x[0] = 5;
x[1] = 4;
x[2] = 3;
```

```

x[3] = 2;
x[4] = 1;

print(x);
"""

execute(program)

[5, 4, 3, 2, 1]

```

Adding to Existing List

`array_name[index_number]`

You can also edit lists after they are declared by adding values at a specified index

`array_name.add(item_to_be_added, index_number)`

There are also shortcuts to add items to the front or back of a list

`array_name.add(item_to_be_added, front)` `array_name.add(item_to_be_added, back)`

```

val program = """
x = [1, 2, 3, 4, 5];
y = ["Hello", "World"];

x.add(10, 0);
x.add(10, 2);
print(x);

y.add("ADDED TO FRONT", front);
y.add("ADDED TO BACK", back);
print(y);
"""

execute(program)

[10, 1, 10, 2, 3, 4, 5]
[ADDED TO FRONT, Hello, World, ADDED TO BACK]

```

List Aggregate Functions

Max

Given a list of integers returns the largest number in the list

`max(array_name);`

Min

Given a list of integers returns the smallest number in the list

```
min(array_name);
```

Sum

Given a list of integers returns the sum of the list

```
sum(array_name);
```

Average

Given a list of integers returns the average of the values in the list

```
avg(array_name);
```

Size

Given any list returns the amount of items inside the list

```
size(array_name);
```

Note: that each of these functions returns a an Integer value

```
val program = """
x = [1, 2, 3, 4, 5];
y = ["Hello", "World"];

print("Max:");
print(max(x));

print("Min:");
print(min(x));

print("Sum:");
print(sum(x));

print("Avg:");
print(avg(x));

print("Size of x:");
print(size(x));
print("Size of y:");
print(size(y));
"""
```

```
execute(program)
```

Max:

5

Min:

1

Sum:

15

Avg:

```
3
Size of x:
5
Size of y:
2
```

Arithmetics

Arithmetics for both strings and integers are supported.

Operations "+" -> Addition for integers | "++" -> Addition for strings "-" -> Subtraction "*" -> Multiplacation "/" -> Division

Integers: can be (1) added, (2) subtracted, (3) multiplied and (4) divided **Strings:** can be (1) added and (2) multiplied

value_one operation *value_two*

Performing arithmetics with an integer will return an integer and performing arithmetics with a string wil return a string

The operations are performed from left to right. However **brackets** can be used to operations

```
val program = """
a = [1, 2, 3, 4, 5];
x = "Hello";
y = "Goodbye";
z = "World";

print(a[0] ++ " " ++ y ++ " " ++ z);
print(a[1] ++ " " ++ (a[1] * (x ++ " ")) ++ z);
"""

execute(program)

1 Goodbye World
2 Hello Hello World
```

If Else Statements

Given a boolean statement perform one block or another

Comparitive Statements: "<" -> Lower than "==" -> Equal ">" -> Greater than

To declare an ifelse statement you must provide a (1) **comparitive statement**, (2) **if_body** and (3) **else_body**. Depending on the boolean value returned by the comparitive statement the if or the else body will be outputed

```
if(comparitive_statement) { if_body
} else { else_body }
```

```

val program = """
x = 10;

if(x < 10) {
    print(x ++ " is lower than 10");
}
else {
    print(x ++ " is not lower than 10");
}
"""

execute(program)

10 is not lower than 10

```

For Loops

Will loop the body of code given a specified amount of time based on the range given. The current iteration of the code will be stored within an integer variable and can be accessed within the body of code. To create a for loop you will need to declare a (1) **Iteration Name**, (2) **Loop Start Number**, (3) **Loop End Number**, (4) **Body of Code**

```
for(Iteration_Name in Loop_Start_Number..Loop_End_Number){ Body_Of_Code }
```

```

val program = """
x = [1, 2, 3, 4, 5];
y = 0;
for(i in 0..4){
    x[i] = 5;
}

print(x);

for(i in 0..4){
    x.add(i, i);
}

print(x);
"""

execute(program)

[5, 5, 5, 5, 5]
[0, 1, 2, 3, 4, 5, 5, 5, 5, 5]

```

Functions

Functions can have as many parameters as needed, but they must stay consistent between function declaration and invocation. When we enter a function the runtime will be copied and

enter its own subscope. Functions in this programming language do not specify a return value, the final value of the function will be treated as a return value

Function Declaration: `function function_name(parameters){ function_body }`

Note that the final expression in the `function_body` is treated as a return value

Function Invocation: `function_name(parameters)`

```
val program = """
function double_n_times(x, n) {
  if(n == 0) {
    x;
  } else {
    double_n_times(2 * x, n - 1);
  }
}

print(double_n_times(10, 3));
"""

execute(program)

80
```

Aggregate Function

Map: Given a list and a function (with parameters if needed) apply the function to each of the values in the list. The return type of the function must match the type of the list given.

`array_name.map(function_name(parameters))`

Note: that the items in the list will automatically be applied as the last parameter inside of the function

```
val program = """
function double_than_add_x(x, y) {
  x + (y * 2);
}

x = [5, 20, 3, 2, 100];
x = x.map(double_than_add_x(1));
print(x);

function greet(x){
  "Hello " ++ x ++ ".";
}

y = ["Bob", "Tim", "Ted", "Larry"];
y = y.map(greet());
print(y);
```

```
function half(x) {  
  x / 2;  
}  
  
z = [2, 4, 6, 8, 10];  
z = sum(z.map(half()));  
print(z);  
"""
```

```
execute(program)
```

```
[11, 41, 7, 5, 201]  
[Hello Bob., Hello Tim., Hello Ted., Hello Larry.]  
15
```