CS 315 - Spring 2023
Homework - 2

First & Last Name: Arda İynem
Department: Computer Science
Section: 01

1. Design issues in each language (Part A)

1.1. Dart

1.1.1. Code Sample

```
String refEnvChecker = "Deep Binding";

void main() {
  int operationNumber = 1;
  var list = [];

  // Operation 1: Are formal and actial parameters type checked?
  print("\nOperation " + operationNumber.toString());
  typeChecking(1, 2);
  // typeChecking(1, "2.0"); ERROR 1: SYNTAX ERROR AT LINE 10
  operationNumber = operationNumber + 1;

  // Operation 2: Are keyword (named) parameters supported?
  print("\nOperation " + operationNumber.toString());
  keywordParameters(1, namedIntVariable: 5);
  keywordParameters(namedIntVariable: 5, 1);
  operationNumber = operationNumber + 1;

  // Operation 3: Are default paremeters supported?
  print("\nOperation " + operationNumber.toString());
  defaultParameters(1, 3);
  defaultParameters(1);
  operationNumber = operationNumber + 1;

  // Operation 4: What are the paremeter passing methods provided?
  print("\nOperation " + operationNumber.toString());
  int tmpInt = 300;

  print("\n-- Before function call --\ntmpInt in main: " +
      tmpInt.toString() +
      "\nlist in main: " +
      list.join(" "));

  parameterPassByValue(list, tmpInt);

  print("\n-- After function call --\ntmpInt in main: " +
      tmpInt.toString() +
```

```
      "\nlist in main: " +
      list.join(" "));
  operationNumber = operationNumber + 1;

  // Operation 5: Can subprograms be passed as parameters? If so, how is the
referencing environment of the passed subprogram bound?
  print("\nOperation " + operationNumber.toString());
  String refEnvChecker = "Ad Hoc Binding";
  subprogramPass(passedFunction, operationNumber);
  operationNumber = operationNumber + 1;
}

void typeChecking(int intVariable, int intVariable2) {
  print((intVariable + intVariable2).toString());
}

void keywordParameters(int intVariable, {required int namedIntVariable}) {
  print((intVariable + namedIntVariable).toString());
}

void defaultParameters(int intVariable, [int defaultIntVariable = 5]) {
  print((intVariable + defaultIntVariable).toString());
}

// Pass-by-value. All variables are reference String, int, double et.
immutable pass-by-assignment. so pass reference by value
void parameterPassByValue(var list, int intVariable) {
  intVariable = 15;
  list.add("addedElement");
  list = ["MutableType_PassByReferenceChecker"];

  print("\n-- In pass-by-value function call --\ntmpInt in function: " +
      intVariable.toString() +
      "\nlist in function: " +
      list.join(" "));
}

void subprogramPass(Function function, int intVariable) {
  String refEnvChecker = "Shallow Binding";
  function(intVariable);
}
```

```
void passedFunction(int intVariable) {
  print("Passed function with a parameter: " + intVariable.toString());
  print("Binding refEnvChecker: " + refEnvChecker.toString());
}
```

1.1.2. Execution Result

```
Operation 1
3

Operation 2
6
6

Operation 3
4
6

Operation 4

-- Before function call --
tmpInt in main: 300
list in main:

-- In pass-by-value function call --
tmpInt in function: 15
list in function: MutableType_PassByReferenceChecker

-- After function call --
tmpInt in main: 300
list in main: addedElement

Operation 5
Passed function with a parameter: 5
Binding refEnvChecker: Deep Binding
```

1.1.2.1. Explanation of Operations

Note: Each operation's corresponding code segment is separated with comments in the sample code. They can be referred to easily so there is no separate Appendix section.

1.1.2.2. Operation 1

Formal parameters and actual parameters are type checked in Dart programming language as the commented out line causes the syntax error *"main.dart:10:20: Error: The argument type 'String' can't be assigned to the parameter type 'int'".* The error

at line 10 indicates the checked types didn’t match and terminated the execution process which is type checking itself.

1.1.2.3. Operation 2

Keyworded (named) parameters are supported in Dart programming language as the two lines of the operation call the function both with positional and named parameters respectively, which outputs the same result. Moreover, Dart does not require named parameters to come after positional parameters as most other programming languages do.

1.1.2.4. Operation 3

Default parameters are supported in Dart programming language as the two lines of the operation call the function both with only one actual parameter (first formal parameter) and with all actual parameters respectively, which outputs different results as first call’s values’ is supplied by both actual parameter and default parameter whereas all parameters of second call is actual parameters.

1.1.2.5. Operation 4

Dart uses only pass-by-value parameter passing method. Therefore, all primitive types are passed by value. However, in Dart, when you pass an object as an argument, you are actually passing a reference to the object since the value of the object variable is the reference address to the object. The reference itself is passed by value, but it still refers to the same object in memory. The modifications to the mutable object's properties can be observed outside the function as a side effect yet reassignments inside the function have no effect outside the function.

To test this passing method an int (primitive type) variable *tmpInt* and a list (non-primitive type) variable *list* are used as actual parameters. The output show that when the actual parameter *tmpInt* is passed into *parameterPassByValue* function formal parameter int variable *intVariable* is initialized with the value of the *tmpInt* which is passed-by-value. Therefore, when *intVariable* is assigned to another int literal, only the value of *intVariable* changes whereas the value of *tmpInt* stays the same which can seen in the output as the before-function and after-function values of tmpInt is same whereas in-function value is different. Similarly, when the actual parameter list variable *list* is assigned to a different list literal the formal parameter *list*’s value (reference address to a list object) doesn’t change. However the line before this reassignment modifies the list object which’s effect can be

seen also after the function. This is because that the list type is a non-primitive type in contrast to int variable which causes side effects on the passed object since the reference address of the object is passed by value and the modifications are done on the object pointed by this address.

In conclusion the underhood mechanism is pass-by-value but the semantics are different that is neither totally pass-by-value nor pass-by-reference because primitive types contain a data value whereas non-primitive types contain reference addresses of the corresponding object in Dart programming language. Hence, this passing methodology semantics is pass-by-value for primitive types and pass-by-object-reference for non-primitive object types

1.1.2.6. Operation 5

Subprogram passing is supported in Dart programming language as the *passedFunction* function which takes an int *intVariable* formal parameter is passed into *subprogramPass* function which takes subprogram (*function)* and int (*intVariable)* formal parameters. Three *refEnvChecker* string variables with the same name and different values are used to check binding of the subprogram. The output shows that the passed subprogram (*passedFunction*) has none of the deep, shallow or ad hoc binding as the *refEnvChecker* variable in the *passedFunction* is bound to *refEnvChecker* variable in the parental scope of the declaration of the subprogram, the case would be different if the programming language had dynamic scoping instead of static (lexical) scoping.

1.2. Go

1.2.1. Code Sample

```
package main

import (
  "fmt"
  "strings"
)
var refEnvChecker = "Deep Binding"

func main() {
  operationNumber := 1
  list := []string{}
```

```
  // Operation 1: Are formal and actial parameters type checked?
  fmt.Println(fmt.Sprintf("\nOperation %v", operationNumber))
  typeChecking(1, 2);
  // typeChecking(1, "2.0"); SYNTAX ERROR AT LINE 16
  operationNumber = operationNumber + 1

  // Operation 2: Are keyword (named) parameters supported?
  // NO
  fmt.Println(fmt.Sprintf("\nOperation %v", operationNumber))
  fmt.Println("NOT SUPPORTED")
  operationNumber = operationNumber + 1

  // Operation 3: Are default paremeters supported?
  // NO
  fmt.Println(fmt.Sprintf("\nOperation %v", operationNumber))
  fmt.Println("NOT SUPPORTED")
  operationNumber = operationNumber + 1

  // Operation 4: What are the paremeter passing methods provided?
  fmt.Println(fmt.Sprintf("\nOperation %v (Pass-by-value)", operationNumber))
  tmpInt := 300

  fmt.Println(fmt.Sprintf("\n-- Before function call --\ntmpInt in main:
%v\nlist in main: %s", tmpInt, strings.Join(list, " ")))

  parameterPassByValue(list, tmpInt);

  fmt.Println(fmt.Sprintf("\n-- After function call --\ntmpInt in main:
%v\nlist in main: %s", tmpInt, strings.Join(list, " ")))

    fmt.Println(fmt.Sprintf("\nOperation %v (Pass-by-reference)",
operationNumber))

  fmt.Println(fmt.Sprintf("\n-- Before function call --\ntmpInt in main:
%v\nlist in main: %s", tmpInt, strings.Join(list, " ")))

  parameterPassByReference(&list, &tmpInt);

  fmt.Println(fmt.Sprintf("\n-- After function call --\ntmpInt in main:
%v\nlist in main: %s", tmpInt, strings.Join(list, " ")))
```

```
  operationNumber = operationNumber + 1

  // Operation 5: Check if a particular element exists in the list
  fmt.Println(fmt.Sprintf("\nOperation %v", operationNumber))
  // refEnvChecker := "Ad Hoc Binding" SYNTAX ERROR AT LINE 53 (NOT USED)
  subprogramPass(passedFunction, operationNumber)
  operationNumber = operationNumber + 1
}

func typeChecking(intVariable int, intVariable2 int) {
  fmt.Println(fmt.Sprintf("%v", (intVariable + intVariable2)))
}



// Pass by value
func parameterPassByValue(list []string, intVariable int) {
  intVariable = 15
  list = append(list, "addedElement")
  list = []string{"MutableType_PassByReferenceChecker"}

  fmt.Println(fmt.Sprintf("\n-- In pass-by-value function call --\ntmpInt in
main: %v\nlist in main: %s", intVariable, strings.Join(list, " ")))
}

// Pass by reference
func parameterPassByReference(list *[]string, intVariable *int) {
  *intVariable = 15
  *list = append(*list, "addedElement")
  *list = []string{"MutableType_PassByReferenceChecker"}

  fmt.Println(fmt.Sprintf("\n-- In pass-by-reference function call --\ntmpInt
in main: %v\nlist in main: %s", *intVariable, strings.Join(*list, " ")))
}

func subprogramPass(function func(intVariable int), intVariable int) {
  // refEnvChecker := "Shallow Binding" SYNTAX ERROR AT LINE 92 (NOT USED)
  function(intVariable)
}

func passedFunction(intVariable int) {
  fmt.Println(fmt.Sprintf("Passed function with a parameter: %v",
intVariable))
```

```
  fmt.Println(fmt.Sprintf("Binding refEnvChecker: %v", refEnvChecker))
}
```

1.2.2. Execution Result

```
Operation 1
3

Operation 2
NOT SUPPORTED

Operation 3
NOT SUPPORTED

Operation 4 (Pass-by-value)

-- Before function call --
tmpInt in main: 300
list in main:

-- In pass-by-value function call --
tmpInt in main: 15
list in main: MutableType_PassByReferenceChecker

-- After function call --
tmpInt in main: 300
list in main:

Operation 4 (Pass-by-reference)

-- Before function call --
tmpInt in main: 300
list in main:

-- In pass-by-reference function call --
tmpInt in main: 15
list in main: MutableType_PassByReferenceChecker

-- After function call --
tmpInt in main: 15
list in main: MutableType_PassByReferenceChecker

Operation 5
Passed function with a parameter: 5
Binding refEnvChecker: Deep Binding
```

1.2.3. Explanation of Operations

Note: Each operation's corresponding code segment is separated with comments in the sample code. They can be referred to easily so there is no separate Appendix section.

1.2.3.1. Operation 1

Formal parameters and actual parameters are type checked in Go programming language as the commented out line causes the syntax error *"main.go:16:19: cannot use "2.0" (untyped string constant) as int value in argument to typeChecking'".* The error at line 16 indicates the checked types didn't match and terminated the execution process which is type checking itself.

1.2.3.2. Operation 2

Keyworded (named) parameters are not supported in Go programming language.

1.2.3.3. Operation 3

Default parameters are not supported in Go programming language

1.2.3.4. Operation 4

Go uses only pass-by-value parameter passing method and supports pass-by-reference semantics with explicit pointers. Therefore, all primitive types are passed by value as well as the non-primitive types which are deep copied to avoid any side effects. However, in Dart, when you pass a pointer to an object as an argument (with &variableName syntax), you are actually passing a reference to the object by passing the value of the pointer which is a reference address. The reference itself is passed by value, but it still refers to the same object in memory. The modifications to any primitive or non-primitive object's properties can be observed outside the function as a side effect yet reassignments inside the function have no effect outside the function.

To test pass-by-value passing method, an int (primitive type) variable *tmpInt* and a slice (non-primitive type) variable *list* are used as actual parameters. The output show that when the actual parameter *tmpInt* is passed into *parameterPassByValue* function formal parameter int variable *intVariable* is initialized with the value of the *tmpInt* which is passed-by-value. Therefore, when *intVariable* is assigned to another int literal, only the value of *intVariable* changes whereas the value of

tmpInt stays the same which can seen in the output as the before-function and after-function values of tmpInt is same whereas in-function value is different. Similarly, when the actual parameter slice variable *list* is assigned to a different slice literal the formal parameter *list*'s value (reference address to a list object) doesn't change. Note that the line before this reassignment modifies the slice object which's effect can not be seen after the function since the slice object is passed by value with deep copying and there is no relation between the slice objects inside and the outside of the function.

To test pass-by-reference semantics, int variable *tmpInt* and slice variable *list* is passed by explicit pointers with & operator before the variable name. Since the explicit pointer arguments contain the reference addresses of the *tmpInt* and *list* the passed value is used to achieve pass-by-reference semantics with pass-by-value method. As expected, outputs show that the modifications and assignments done on the objects persist after the function.

In conclusion the underhood mechanism is pass-by-value but explicit pointers allow to achieve pass-by-reference semantics with the pass-by-value method in Go programming language.

1.2.3.5. Operation 5

Subprogram passing is supported in Go programming language as the *passedFunction* function which takes an int *intVariable* formal parameter is passed into *subprogramPass* function which takes subprogram (*function)* and int (*intVariable)* formal parameters. Three *refEnvChecker* string variables with the same name and different values are used to check binding of the subprogram. The output shows that the passed subprogram (*passedFunction*) has **deep binding** as the *refEnvChecker* variable in the *passedFunction* is bound to *refEnvChecker* variable in the parental scope of the declaration of the subprogram as other commented out *refEnvChecker* variables causes syntax error("*./main.go:83:3: refEnvChecker declared but not used")* as they are never used, the case would be different if the programming language had dynamic scoping instead of static (lexical) scoping.

1.3. Javascript

1.3.1. Code Sample

```
<script>
```

```
var refEnvChecker = "Deep Binding";

function main() {
  var operationNumber = 1;
  var list = [];

  // Operation 1: Are formal and actial parameters type checked?
  console.log("\nOperation " + operationNumber);
  typeChecking(1, 2);
  typeChecking(1, "2.0"); // NO TYPE CHECKING
  operationNumber = operationNumber + 1;

  // Operation 2: Are keyword (named) parameters supported?
  // No, but object destructring
  console.log("\nOperation " + operationNumber.toString());
  console.log("NOT SUPPORTED");
  operationNumber = operationNumber + 1;

  // Operation 3: Are default paremeters supported?
  console.log("\nOperation " + operationNumber.toString());
  defaultParameters(1, 3);
  defaultParameters(1);
  operationNumber = operationNumber + 1;

  // Operation 4: What are the paremeter passing methods provided?
  console.log("\nOperation " + operationNumber.toString());

  var tmpInt = 300;

  console.log("\n-- Before function call --\ntmpInt in main: " +
    tmpInt.toString() +
    "\nlist in main: " +
    list.join(" "));

  parameterPassByValue(list, operationNumber);

  console.log("\n-- After function call --\ntmpInt in main: " +
    tmpInt.toString() +
    "\nlist in main: " +
    list.join(" "));
  operationNumber = operationNumber + 1;
```

```
    // Operation 5: Can subprograms be passed as parameters? If so, how is the
referencing environment of the passed subprogram bound?
    console.log("\nOperation " + operationNumber.toString());
    var refEnvChecker = "Ad Hoc Binding";
    subprogramPass(passedFunction, operationNumber);
    operationNumber = operationNumber + 1;
  }

  function typeChecking(intVariable, intVariable2) {
    console.log((intVariable + intVariable2).toString());
  }

  function defaultParameters(intVariable, defaultIntVariable = 5) {
    console.log((intVariable + defaultIntVariable).toString());
  }

  // Only Pass-by-value where all objects' values are references Therefore it
is pass-reference-by-value
  function parameterPassByValue(list, intVariable) {
    intVariable = 15;
    list.push("addedElement");

    console.log("\n-- In pass-by-value function call --\ntmpInt in function: "
+
      intVariable.toString() +
      "\nlist in function: " +
      list.join(" "));
  }

  function subprogramPass(functionVariable, intVariable) {
    var refEnvChecker = "Shallow Binding";
    functionVariable(intVariable);
  }

  function passedFunction(intVariable) {
    console.log("Passed function with a parameter: " +
intVariable.toString());
    console.log("Binding refEnvChecker: " + refEnvChecker.toString());
  }

  main();
```

```
</script>
```

1.3.2. Execution Result

```
"
Operation 1"
"3"
"12.0"
"
Operation 2"
"NOT SUPPORTED"
"
Operation 3"
"4"
"6"
"
Operation 4"
"
-- Before function call --
tmpInt in main: 300
list in main: "
"
-- In pass-by-value function call --
tmpInt in function: 15
list in function: addedElement"
"
-- After function call --
tmpInt in main: 300
list in main: addedElement"
"
Operation 5"
"Passed function with a parameter: 5"
"Binding refEnvChecker: Deep Binding"
```

1.3.3. Explanation of Operations

Note: Each operation's corresponding code segment is separated with comments in the sample code. They can be referred to easily so there is no separate Appendix section.

1.3.3.1. Operation 1

Formal parameters and actual parameters are not type checked in Javascript programming language as Javascript has dynamic type binding, actual arguments are accepted

without any type checking and parameters are coerced to each other if it is possible. The two lines related to operation and their output shows this behavior.

1.3.3.2. Operation 2

Keyworded (named) parameters are not supported in Javascript programming language.

1.3.3.3. Operation 3

Default parameters are supported in Javascript programming language as the two lines of the operation call the function both with only one actual parameter (first formal parameter) and with all actual parameters respectively, which outputs different results as first call's values' is supplied by both actual parameter and default parameter whereas all parameters of second call is actual parameters.

1.3.3.4. Operation 4

Javascript uses only pass-by-value parameter passing method. Therefore, all primitive types are passed by value. However, in Javascript , when you pass an object as an argument, you are actually passing a reference to the object since the value of the object variable is the reference address to the object. The reference itself is passed by value, but it still refers to the same object in memory. The modifications to the mutable object's properties can be observed outside the function as a side effect yet reassignments inside the function have no effect outside the function.

To test this passing method an int (primitive type) variable *tmpInt* and a array (non-primitive type) variable *list* are used as actual parameters. The output show that when the actual parameter *tmpInt* is passed into *parameterPassByValue* function formal parameter int variable *intVariable* is initialized with the value of the *tmpInt* which is passed-by-value. Therefore, when *intVariable* is assigned to another int literal, only the value of *intVariable* changes whereas the value of *tmpInt* stays the same which can seen in the output as the before-function and after-function values of tmpInt is same whereas in-function value is different. Similarly, when the actual parameter array variable *list* is assigned to a different array literal the formal parameter *list*'s value (reference address to an array object) doesn't change. However the line before this reassignment modifies the array object which's effect can be seen also after the function. This is because that the array type is a non-primitive type in contrast to int variable

which causes side effects on the passed object since the reference address of the object is passed by value and the modifications are done on the object pointed by this address.

In conclusion the underhood mechanism is pass-by-value but the semantics are different that is neither totally pass-by-value nor pass-by-reference because primitive types contain a data value whereas non-primitive types contain reference addresses of the corresponding object in Javascript programming language. Hence, this passing methodology semantics is pass-by-value for primitive types and pass-by-object-reference for non-primitive object types

1.3.3.5. Operation 5

Subprogram passing is supported in Javascript programming language as the *passedFunction* function which takes an int *intVariable* formal parameter is passed into *subprogramPass* function which takes subprogram (*functionVariable)* and int (*intVariable)* formal parameters. Three *refEnvChecker* string variables with the same name and different values are used to check binding of the subprogram. The output shows that the passed subprogram (*passedFunction*) has **deep binding** as the *refEnvChecker* variable in the *passedFunction* is bound to *refEnvChecker* variable in the parental scope of the declaration of the subprogram, the case would be different if the programming language had dynamic scoping instead of static (lexical) scoping.

1.4. Lua

1.4.1. Code Sample

```
refEnvChecker = "Deep Binding"

function main()
  local operationNumber = 1
  local list = {}

  -- Operation 1: Are formal and actial parameters type checked?
  print("\nOperation " .. operationNumber)
  typeChecking(1, 2)
  typeChecking(1, "2.0") -- NO TYPE CHECKING
  operationNumber = operationNumber + 1

  -- Operation 2: Are keyword (named) parameters supported?
```

```
  -- NO
  print("\nOperation " .. operationNumber)
  print("NOT SUPPORTED")
  operationNumber = operationNumber + 1

  -- Operation 3: Are default paremeters supported?
  -- NO
  print("\nOperation " .. operationNumber)
  print("NOT SUPPORTED")
  operationNumber = operationNumber + 1

  -- Operation 4: What are the paremeter passing methods provided?
  print("\nOperation " .. operationNumber)
  local tmpInt = 300

  print("\n-- Before function call --\ntmpInt in main: " ..
    tmpInt .. "\nlist in main: " ..
    table.concat(list, " "))

  parameterPassByValue(list, tmpInt)

  print("\n-- After function call --\ntmpInt in main: " ..
    tmpInt .. "\nlist in main: " ..
    table.concat(list, " "))

  operationNumber = operationNumber + 1

  -- Operation 5: Can subprograms be passed as parameters? If so, how is the
referencing environment of the passed subprogram bound?
  print("\nOperation " .. operationNumber)
  local refEnvChecker = "Ad Hoc Binding"
  functionVariable = passedFunction
  subprogramPass(functionVariable, operationNumber)
  operationNumber = operationNumber + 1

end

function typeChecking(intVariable, intVariable2)
  print(intVariable + intVariable2)
end
```

```lua
-- all types are passed by value, but function, table, userdata and thread are reference (coroutine) types and are (imitating) passed by reference (shallow copied passed by value).
function parameterPassByValue(list, intVariable)
  intVariable = 15
  table.insert(list, "addedElement")
  list = {"MutableType_PassByReferenceChecker"}

  print("\n-- In pass-by-value function call --\ntmpInt in main: " ..
    intVariable .. "\nlist in main: " ..
    table.concat(list, " "))
end

function subprogramPass(functionVariable, intVariable)
  local refEnvChecker = "Shallow Binding"
  functionVariable(intVariable)
end

function passedFunction(intVariable)
  print("Passed function with a parameter: " .. intVariable)
  print("Binding refEnvChecker: " .. refEnvChecker)
end

main()
```

1.4.2. Execution Result

```
Operation 1
3
3

Operation 2
NOT SUPPORTED

Operation 3
NOT SUPPORTED

Operation 4

-- Before function call --
tmpInt in main: 300
list in main:

-- In pass-by-value function call --
tmpInt in main: 15
list in main: MutableType_PassByReferenceChecker

-- After function call --
tmpInt in main: 300
list in main: addedElement

Operation 5
Passed function with a parameter: 5
Binding refEnvChecker: Deep Binding
```

1.4.3. Explanation of Operations

Note: Each operation's corresponding code segment is separated with comments in the sample code. They can be referred to easily so there is no separate Appendix section.

1.4.3.1. Operation 1

Formal parameters and actual parameters are not type checked in Lua programming language as Lua has dynamic type binding, actual arguments are accepted without any type checking and parameters are coerced to each other if it is possible. The two lines related to operation and their output shows this behavior.

1.4.3.2. Operation 2

Keyworded (named) parameters are not supported in the Lua programming language.

1.4.3.3. Operation 3

Default parameters are not supported in Lua programming language.

1.4.3.4. Operation 4

Lua uses only pass-by-value parameter passing method. Therefore, all primitive types are passed by value. However, in Lua , when you pass an object (tables, functions, threads, and userdata values) as an argument, you are actually passing a reference to the object since the value of the object variable is the reference address to the object. The reference itself is passed by value, but it still refers to the same object in memory. The modifications to the mutable object's properties can be observed outside the function as a side effect yet reassignments inside the function have no effect outside the function.

To test this passing method an int (primitive type) variable *tmpInt* and a table (non-primitive type) variable *list* are used as actual parameters. The output show that when the actual parameter *tmpInt* is passed into *parameterPassByValue* function formal parameter int variable *intVariable* is initialized with the value of the *tmpInt* which is passed-by-value. Therefore, when *intVariable* is assigned to another int literal, only the value of *intVariable* changes whereas the value of *tmpInt* stays the same which can seen in the output as the before-function and after-function values of tmpInt is same whereas in-function value is different. Similarly, when the actual parameter table variable *list* is assigned to a different table literal the formal parameter *list*'s value (reference address to an table object) doesn't change. However the line before this reassignment modifies the table object which's effect can be seen also after the function. This is because that the table type is a non-primitive type in contrast to int variable which causes side effects on the passed object since the reference address of the object is passed by value and the modifications are done on the object pointed by this address.

In conclusion the underhood mechanism is pass-by-value but the semantics are different that is neither totally pass-by-value nor pass-by-reference because primitive types contain a data value whereas non-primitive types contain reference addresses of the corresponding object in Lua programming

language. Hence, this passing methodology semantics is pass-by-value for primitive types and pass-by-object-reference for non-primitive object types

1.4.3.5. Operation 5

Subprogram passing is supported in Lua programming language as the *passedFunction* function which takes an int *intVariable* formal parameter is passed into *subprogramPass* function which takes subprogram (*functionVariable)* and int (*intVariable)* formal parameters. Three *refEnvChecker* string variables with the same name and different values are used to check binding of the subprogram. The output shows that the passed subprogram (*passedFunction*) has **deep binding** as the *refEnvChecker* variable in the *passedFunction* is bound to *refEnvChecker* variable in the parental scope of the declaration of the subprogram, the case would be different if the programming language had dynamic scoping instead of static (lexical) scoping.

1.5. Python

1.5.1. Code Sample

```
refEnvChecker = "Deep Binding"

def main():
  operationNumber = 1
  list = []

  # Operation 1: Are formal and actial parameters type checked?
  print("\nOperation " + str(operationNumber))
  typeChecking(1, 2)
  # typeChecking(1, "2.0") SYNTAX ERROR AT LINE 46 NOT ON 10 SO NO TYPE
CHECKING
  operationNumber = operationNumber + 1

  # Operation 2: Are keyword (named) parameters supported?
  print("\nOperation " + str(operationNumber))
  keywordParameters(1, namedIntVariable=5)
  keywordParameters(namedIntVariable=5, intVariable=1)
  # keywordParameters(namedIntVariable=5, 1) Positional arguements after
named, sytnax error
  operationNumber = operationNumber + 1
```

```
  # Operation 3: Are default paremeters supported?
  print("\nOperation " + str(operationNumber))
  defaultParameters(1, 3)
  defaultParameters(1)
  operationNumber = operationNumber + 1

  # Operation 4: What are the paremeter passing methods provided?
  print("\nOperation " + str(operationNumber))
  tmpInt = 300

  print("\n-- Before function call --\ntmpInt in main: " + str(tmpInt) +
        "\nlist in main: " + " ".join(list))

  parameterPassByAssignment(list, operationNumber)

  print("\n-- After function call --\ntmpInt in main: " + str(tmpInt) +
        "\nlist in main: " + " ".join(list))

  operationNumber = operationNumber + 1

  # Operation 5: Can subprograms be passed as parameters? If so, how is the
referencing environment of the passed subprogram bound?
  print("\nOperation " + str(operationNumber))
  refEnvChecker = "Ad Hoc Binding"
  subprogramPass(passedFunction, operationNumber)
  operationNumber = operationNumber + 1


def typeChecking(intVariable, intVariable2):
  print(str(intVariable + intVariable2))


def keywordParameters(intVariable, namedIntVariable):
  print(str(intVariable + namedIntVariable))


def defaultParameters(intVariable, defaultIntVariable=5):
  print(str(intVariable + defaultIntVariable))
```

```
# In python all variables are references to objects so the value of the
variables are passed which are references. a.k.a pass-reference-by-value,
a.k.a passbyassignment. Immutable types does not change whereas mutable
change. pass-by-assignment is in effect pass-by- reference, because the value
of all actual parameters are references (IN BOOK)
def parameterPassByAssignment(list, intVariable):
  intVariable = 15
  list.append("addedElement")
  list = ["MutableType_PassByReferenceChecker"]

  print("\n-- In pass-by-assignment function call --\ntmpInt in main: " +
        str(intVariable) + "\nlist in main: " + " ".join(list))

def subprogramPass(function, intVariable):
  refEnvChecker = "Shallow Binding"
  function(intVariable)

def passedFunction(intVariable):
  print("Passed function with a parameter: " + str(intVariable))
  print("Binding refEnvChecker: " + str(refEnvChecker))

main()
```

1.5.2. Execution Result

```
Operation 1
3

Operation 2
6
6

Operation 3
4
6

Operation 4

-- Before function call --
tmpInt in main: 300
list in main:

-- In pass-by-assignment function call --
tmpInt in main: 15
list in main: MutableType_PassByReferenceChecker
```

```
-- After function call --
tmpInt in main: 300
list in main: addedElement

Operation 5
Passed function with a parameter: 5
Binding refEnvChecker: Deep Binding
```

1.5.3. Explanation of Operations

Note: Each operation's corresponding code segment is separated with comments in the sample code. They can be referred to easily so there is no separate Appendix section.

1.5.3.1. Operation 1

Formal parameters and actual parameters are not type checked in the Python programming language. However, the commented out line causes the syntax error *"TypeError: unsupported operand type(s) for +: 'int' and 'str'"* which is on another line (line 49) and not on line 10. This also shows that there is no type checking as actual parameters are accepted no matter what their types and these accepted -but not suitable- types triggered an error in the following lines.

1.5.3.2. Operation 2

Keyworded (named) parameters are supported in Python programming language as the two lines of the operation call the function both with positional and named parameters respectively, which outputs the same result. However, the commented out third line shows that Python strictly requires named parameters to come after positional parameters. Furthermore, Python directly takes the variable name as parameter keyword so it is flexible that any variable can behave like either named argument or positional argument without any specification required.

1.5.3.3. Operation 3

Default parameters are supported in Python programming language as the two lines of the operation call the function both with only one actual parameter (first formal parameter) and with all actual parameters respectively, which outputs different results as first call's values' is supplied by both actual

parameter and default parameter whereas all parameters of second call is actual parameters.

1.5.3.4. Operation 4

In theory Python uses only pass-by-value parameter passing method. However, in Python all variables are references to objects (data values), which results in variables passing the reference data they hold as a value when the programming language uses pass-by-value. So when the reference data is copied with pass-by-value methodology the reference data can be used to access the object itself which can be modified if it is a mutable (modifiable) type. Yet, when the copied reference is assigned again to another variable in the function, only the temporarily created variable's value (reference address of an object) changes thus the outer variable will still be containing the old object's reference address as a value. This passing method can be named as **pass-by-assignment (in Python terminology)** or **pass-by-object-reference** even though it has pass-by-value in its essence.

To test this passing method an int (immutable type) variable *tmpInt* and a list (mutable type) variable *list* are used as actual parameters. The output show that when the actual parameter *tmpInt* is passed into *parameterPassByAssignment* function formal parameter int variable *intVariable* is initialized with the value of the *tmpInt* which is passed-by-value. Therefore, when *intVariable* is assigned to another int literal, only the value (reference address of an int object) of *intVariable* changes whereas the value of *tmpInt* stays the same which can seen in the output as the before-function and after-function values of tmpInt is same whereas in-function value is different. Similarly, when the actual parameter list variable *list* is assigned to a different list literal the formal parameter *list*'s value doesn't change. However the line before this reassignment modifies the list object which's effect can be seen also after the function. This is because that the list type is mutable in contrast to int variable which causes side effects on the passed object since the reference address of the object is passed by value and the modifications are done on the object pointed by this address.

In conclusion the underhood mechanism is pass-by-value but the semantics are different that is neither totally pass-by-value nor pass-by-reference because all data values are objects and variables contain reference addresses of these objects in Python programming language. Hence, this passing methodology semantics is called by pass-by-assignment (in Python terminology) or pass-by-object-reference.

1.5.3.5. Operation 5

Subprogram passing is supported in Python programming language as the *passedFunction* function which takes an int *intVariable* formal parameter is passed into *subprogramPass* function which takes subprogram (*function)* and int (*intVariable)* formal parameters. Three *refEnvChecker* string variables with the same name and different values are used to check binding of the subprogram. The output shows that the passed subprogram (*passedFunction*) has **deep binding** as the *refEnvChecker* variable in the *passedFunction* is bound to *refEnvChecker* variable in the parental scope of the declaration of the subprogram, the case would be different if the programming language had dynamic scoping instead of static (lexical) scoping.

1.6. Ruby

1.6.1. Code Sample

```
refEnvChecker = "Deep Binding"

def main
  operationNumber = 1
  list = []

  # Operation 1: Are formal and actial parameters type checked?
  puts "\nOperation " + operationNumber.to_s
  typeChecking(1, 2)
  # typeChecking(1, "2.0") SYNTAX ERROR AT LINE 51 NOT ON 10 SO NO TYPE
CHECKING
  operationNumber += 1

  # Operation 2: Are keyword (named) parameters supported?
  puts "\nOperation " + operationNumber.to_s
  keywordParameters(1, namedIntVariable:5)
  # keywordParameters(namedIntVariable:5, 1) SYNTAX ERROR POSITIONAL
PARAMETERS AFTER NAMED
  operationNumber += 1

  # Operation 3: Are default paremeters supported?
  puts "\nOperation " + operationNumber.to_s
  defaultParameters(1, 3)
  defaultParameters(1)
```

```ruby
  operationNumber += 1

  # Operation 4: What are the paremeter passing methods provided?
  puts "\nOperation " + operationNumber.to_s
  tmpInt = 300

  puts "\n-- Before function call function call --\nintVariable in function: "
+
       tmpInt.to_s +
       "\nlist in function: " +
       list.join(' ')

  parameterPassByValue(list, operationNumber)

  puts "\n-- After function call --\nintVariable in function: " +
       tmpInt.to_s +
       "\nlist in function: " +
       list.join(' ')

  operationNumber += 1

  # Operation 5: Can subprograms be passed as parameters? If so, how is the
referencing environment of the passed subprogram bound?
  puts "\nOperation " + operationNumber.to_s
  refEnvChecker = "Ad Hoc Binding"
  subprogramPass(method(:passedFunction), operationNumber)
  operationNumber += 1
end

def typeChecking(intVariable, intVariable2)
  puts (intVariable + intVariable2).to_s
end

def keywordParameters(intVariable, namedIntVariable:)
  puts (intVariable + namedIntVariable).to_s
end

def defaultParameters(intVariable, defaultIntVariable = 5)
  puts (intVariable + defaultIntVariable).to_s
end
```

```ruby
# Ruby uses only pass-by-value however all variables are references so this
pass-by-value works like a pass-reference-by-value. Immutable types don't
change like int, double pass-by-assignment
def parameterPassByValue(list, intVariable)
  intVariable = 15
  list.push("addedElement")
  list = ["MutableType_PassByReferenceChecker"]

  puts "\n-- In pass-by-value function call --\nintVariable in function: " +
       intVariable.to_s +
       "\nlist in function: " +
       list.join(' ')
end

def subprogramPass(callback, intVariable)
  refEnvChecker = "Shallow Binding"
  callback.call(intVariable)
end

def passedFunction(intVariable)
  puts "Passed function with a parameter: " + (intVariable).to_s
  # puts "Binding refEnvChecker: " + (refEnvChecker).to_s SYNTAX ERROR
undefined local variable or method `refEnvChecker'
end

main
```

1.6.2. Execution Result

```
Operation 1
3

Operation 2
6

Operation 3
4
6

Operation 4

-- Before function call function call --
intVariable in function: 300
list in function:

-- In pass-by-value function call --
intVariable in function: 15
list in function: MutableType_PassByReferenceChecker

-- After function call --
intVariable in function: 300
list in function: addedElement

Operation 5
Passed function with a parameter: 5
```

1.6.3. Explanation of Operations

Note: Each operation's corresponding code segment is separated with comments in the sample code. They can be referred to easily so there is no separate Appendix section.

1.6.3.1. Operation 1

Formal parameters and actual parameters are not type checked in the Ruby programming language. However, the commented out line causes the syntax error *"main.rb:52:in `+': String can't be coerced into Integer (TypeError)"* which is on another line (line 52) and not on line 10. This also shows that there is no type checking as actual parameters are accepted no matter what their types and these accepted -but not suitable- types triggered an error in the following lines.

1.6.3.2. Operation 2

Keyworded (named) parameters are supported in Ruby programming language as the first line of the operation calls the function both with positional and named parameters respectively, which outputs the same result. However, the commented out second line shows that Python strictly requires named parameters to come after positional parameters.

1.6.3.3. Operation 3

The array object is passed into the *isListEmpty* function which accepts an array variable. *isListEmpty* function that I defined uses the built-in *empty?* property of arrays returning a bool value depending on the emptiness of the array. Then the corresponding result is printed within an if-else block.

1.6.3.4. Operation 4

In theory Ruby uses only pass-by-value parameter passing method. However, in Ruby all variables are references to objects (data values), which results in variables passing the reference data they hold as a value when the programming language uses pass-by-value. So when the reference data is copied with pass-by-value methodology the reference data can be used to access the object itself which can be modified if it is a mutable (modifiable) type. Yet, when the copied reference is assigned again to another variable in the function, only the temporarily created variable's value (reference address of an object) changes thus the outer variable will still be containing the old object's reference address as a value. This passing method can be named as **pass-by-assignment (in Python terminology)** or **pass-by-object-reference** even though it has pass-by-value in its essence.

To test this passing method an int (immutable type) variable *tmpInt* and a array (mutable type) variable *list* are used as actual parameters. The output show that when the actual parameter *tmpInt* is passed into *parameterPassByValue* function formal parameter int variable *intVariable* is initialized with the value of the *tmpInt* which is passed-by-value. Therefore, when *intVariable* is assigned to another int literal, only the value (reference address of an int object) of *intVariable* changes whereas the value of *tmpInt* stays the same which can seen in the output as the before-function and after-function values of tmpInt is same whereas in-function value is different. Similarly, when the actual parameter array variable *list* is assigned to a different array literal the formal parameter *list*'s value doesn't change. However the line before this reassignment modifies the array object which's effect can

be seen also after the function. This is because that the array type is mutable in contrast to int variable which causes side effects on the passed object since the reference address of the object is passed by value and the modifications are done on the object pointed by this address.

In conclusion the underhood mechanism is pass-by-value but the semantics are different that is neither totally pass-by-value nor pass-by-reference because all data values are objects and variables contain reference addresses of these objects in Ruby programming language. Hence, this passing methodology semantics is called by pass-by-assignment (in Python terminology) or pass-by-object-reference.

1.6.3.5. Operation 5

Subprogram passing is supported in Ruby programming language as the *passedFunction* function which takes an int *intVariable* formal parameter is passed into *subprogramPass* function which takes subprogram (*function)* and int (*intVariable)* formal parameters. Three *refEnvChecker* string variables with the same name and different values are used to check binding of the subprogram. The output shows that the passed subprogram (*passedFunction*) has none of the deep, shallow or ad hoc binding as the *refEnvChecker* variable in the *passedFunction* causes the syntax error (“*main.rb:82:in `passedFunction': undefined local variable or method `refEnvChecker' for main:Object (NameError)”).* This is because local variables are tightly scoped in Ruby and the function cannot access outer variables if they are not passed as arguments.

1.7. Rust

1.7.1. Code Sample

```rust
static refEnvChecker: &'static str = "Deep Binding";

fn main() {
    let mut operation_number: i32 = 1;
    let mut list: Vec<String> = vec![];

    // Operation 1: Are formal and actual parameters type checked?
    println!("\nOperation {}", operation_number);
    type_checking(1, 2);
    // type_checking(1, "2.0"); SYNTAX ERROR AT LINE 10
```

```
    operation_number += 1;

    // Operation 2: Are keyword (named) parameters supported?
    // No
    println!("\nOperation {}", operation_number);
    println!("Not Supported");
    operation_number += 1;

    // Operation 3: Are default parameters supported?
    // No
    println!("\nOperation {}", operation_number);
    println!("Not Supported");
    operation_number += 1;

    // Operation 4: What are the parameter passing methods provided?
    println!("\nOperation {} (Pass by Move)", operation_number);

    let mut tmp_int = 300;

    println!(
        "\n-- Before function call --\ntmp_int in main: {}\nlist in main: {}",
        tmp_int,
        list.join(" ")
    );

    // parameter_pass_by_value(list, tmp_int); passes the ownership which is
moved to function back to the list variable to make it accessible again
    list = parameter_pass_move(list, tmp_int);

    println!(
        "\n-- After function call --\ntmp_int in main: {}\nlist in main: {}",
        tmp_int,
        list.join(" ")
    );

    println!("\nOperation {} (Pass by Borrow)", operation_number);

    println!(
        "\n-- Before function call --\ntmp_int in main: {}\nlist in main: {}",
        tmp_int,
        list.join(" ")
    );
```

```
    parameter_pass_borrow(&mut list, &mut tmp_int);

    println!(
        "\n-- After function call --\ntmp_int in main: {}\nlist in main: {}",
        tmp_int,
        list.join(" ")
    );

    operation_number += 1;

    // Operation 5: Can subprograms be passed as parameters? If so, how is the
referencing environment of the passed subprogram bound?
    println!("\nOperation {}", operation_number);
    // let refEnvChecker: &str = "Ad Hoc Binding"; SYNTAX ERROR WHEN GLOBAL
REFENVCHECKER IS COMMENTED OUT
    subprogramPass(passedFunction, operation_number);
    operation_number += 1;
}

fn type_checking(int_variable: i32, int_variable2: i32) {
    println!("{}", int_variable + int_variable2);
}

fn parameter_pass_move(mut list: Vec<String>, mut int_variable: i32) ->
Vec<String> {
    int_variable = 15;
    list.push(String::from("addedElement"));
    list = vec!["MutableType_MoveChecker".to_string()];
    // LIST VARIABLE CAN NOT BE USED FOR PASS BY VALUE AS IT DOES NOT HAVE A
BUILT IN COPY TRAIT

    println!(
        "\n-- In pass-move function call --\ntmp_int in main: {}\nlist in
main: {}",
        int_variable,
        list.join(" ")
    );

    list
}
```

```rust
fn parameter_pass_borrow(list: &mut Vec<String>, int_variable: &mut i32) {
    *int_variable = 15;
    (*list).push(String::from("addedElement"));

    *list = vec!["MutableType_BorrowChecker".to_string()];

    println!(
        "\n-- In pass-borrow function call --\ntmp_int in main: {}\nlist in
main: {}",
        int_variable,
        list.join(" ")
    );
}

fn subprogramPass(function: fn(i32), int_variable: i32) {
    // let refEnvChecker: &str = "Ad Hoc Binding"; SYNTAX ERROR WHEN GLOBAL
REFENVCHECKER IS COMMENTED OUT
    function(int_variable);
}

fn passedFunction(int_variable: i32) {
    println!("Passed function with a parameter: {}", int_variable);
    println!("Binding refEnvChecker: {}", refEnvChecker);
}
```

1.7.2. Execution Result

```
Operation 1
3

Operation 2
Not Supported

Operation 3
Not Supported

Operation 4 (Pass by Move)

-- Before function call --
tmp_int in main: 300
list in main:
```

```
-- In pass-move function call --
tmp_int in main: 15
list in main: MutableType_MoveChecker

-- After function call --
tmp_int in main: 300
list in main: MutableType_MoveChecker

Operation 4 (Pass by Borrow)

-- Before function call --
tmp_int in main: 300
list in main: MutableType_MoveChecker

-- In pass-borrow function call --
tmp_int in main: 15
list in main: MutableType_BorrowChecker

-- After function call --
tmp_int in main: 15
list in main: MutableType_BorrowChecker

Operation 5
Passed function with a parameter: 5
Binding refEnvChecker: Deep Binding
```

1.7.3. Explanation of Operations

Note: Each operation's corresponding code segment is separated with comments in the sample code. They can be referred to easily so there is no separate Appendix section.

1.7.3.1. Operation 1

Formal parameters and actual parameters are type checked in Rust programming language as the commented out line causes the syntax error "*error[E0308]: mismatched types*". The error at line 10 indicates the checked types didn't match and terminated the execution process which is type checking itself.

1.7.3.2. Operation 2

Keyworded (named) parameters are not supported in Rust programming language.

1.7.3.3. Operation 3

Default parameters are not supported in Rust programming language

1.7.3.4. **Operation 4**

Rust uses a parameter passing mechanism known as "move semantics" or "move-by-default." When you pass arguments to a function in Rust, the ownership of the variables is transferred to the function. This means that the original variables are moved into the function, and they are no longer accessible in the calling scope unless they are explicitly returned.

However, Rust also provides the concept of "references" and "borrowing." By using references (with &variableName for immutable variables and &mut variableName for mutable variables as syntax), you can pass a reference to a value instead of moving ownership. References allow borrowing the value temporarily without taking ownership, and they enable functions to read or modify the borrowed value without transferring ownership. Moreover, the types that implement the Copy trait (mostly basic types such as integers, booleans, etc.) are deep copied with pass-by-value method instead of transferring their ownership.

To test move-by-default passing methods for both pass-by-value and pass-by-reference semantics, an i32 (primitive type) variable *tmp_int* and a vector (non-primitive type) variable *list* are used as actual parameters. The output shows that when the actual parameter *tmp_int* is passed into *parameter_pass_by_value* function formal parameter i32 variable *int_variable* is initialized with the value of the *tmp_int* which is passed-by-value. Therefore, when *int_variable* is assigned to another i32 literal, only the value of *int_variable* changes whereas the value of *tmp_int* stays the same which can be seen in the output as the before-function and after-function values of *tmp_int* is same whereas in-function value is different. However, when the actual parameter vector variable *list* is assigned to the formal parameter, the data value is completely moved into the function as vector type does not have built-in *Copy* trait and with this parameter passing its ownership and data is moved into the calling function which can be only returned by returning it from the function while assigning it to variable outside the function again. Note that reassignment modifies the vector object which’s effect can be seen also after the function which exhibits pass-by-reference semantics. This is because the vector type does not have built-in copy trait to have pass-by-value semantics in contrast to the i32 type.

To test borrowing passing methods for pass-by-reference semantics, an i32 (primitive type) variable *tmp_int* and a vector

(non-primitive type) variable *list* are used as actual parameters. The output shows that when the actual parameter *tmp_int* is passed into *parameter_pass_by_value* function formal parameter i32 variable *int_variable* is passed the reference of the *tmp_int* which is passed-by-reference semantics in common sense and borrowing in Rust sense which means a temporary move (which will be moved back after function in contrast to move-by-default semantics) by passing the reference addresses. Therefore, when *int_variable* is assigned to another i32 literal (after dereferenced with * operator before variable name), the value of *int_variable* changes meaning that the value of *tmp_int* will also be changed after the borrowed variable is returned at the end of function which can be seen in the output as the in-function and after-function values of *tmp_int* is same whereas before-function value is different. Similarly, when the actual parameter vector variable *list* is borrowed into the formal parameter vector variable *list* the variable is borrowed and the reference address is passed into function which is then deference to modify, access and reassign which is directly the pass-by-reference semantics. The output shows this behavior and how pass-by-borrow is used to achieve pass-by-reference semantics.

To summarize, Rust primarily uses move semantics for parameter passing, transferring ownership of variables to functions which results in both pass-by-value and pass-by-reference semantics. It also provides references and borrowing as an alternative mechanism for working with values without transferring ownership which results in pass-by-reference semantics.

1.7.3.5. **Operation 5**

Subprogram passing is supported in Rust programming language as the *passedFunction* function which takes an i32 *int_variable* formal parameter is passed into *subprogramPass* function which takes subprogram (*function)* and int (*int_variable)* formal parameters. Three *refEnvChecker* string variables with the same name and different values are used to check binding of the subprogram. The output shows that the passed subprogram (*passedFunction*) has **deep binding** as the *refEnvChecker* variable in the *passedFunction* is bound to *refEnvChecker* variable in the parental scope of the declaration of the subprogram and the other *refEnvChecker* variables causes syntax errors (“*error[E0425]: cannot find value `refEnvChecker` in this scope*”) as they are not in the scope of the subprogram, the case would be different if the

programming language had dynamic scoping instead of static (lexical) scoping.

2. Evaluation (Part B)

Note: Since only list operations of the language are evaluated in terms of **Readability** and **Writability** of the programming languages; **Simplicity, Orthogonality,** and **Syntax Design** can only be superficially evaluated and commented on as language evaluation criteria due to narrow usage of the languages. Therefore these criteria are not directly referred to in evaluation process yet they are considered as a whole when categorizing the evaluation criteria below:

Evaluation Criteria

Type Checking

Parameter Type Checking (+1 Points)

- Readability (+2 Points)
  - It is easier to guess what type the parameters are, especially when the code amount gets higher complexity is avoided.
  - Searching for a bug caused by a type mistake is easier
- Writability (-1 Point)
  - The necessity to explicitly specify which type are the variables is prevented which saves significant effort for the developer.
- Total Point → 2 - 1 = +1

No Parameter Type Checking (-1 Points)

Since the reasons and points are the opposite of Parameter Type Checking Criterion, they are not specified to avoid repetition

Keyword (Named) Arguments

Named Arguments (+1 Points)

- Readability (+2 Points)
  - It is easier to understand the relation between the values and parameter variables while passing arguments to a function.
  - Positional flexibility of named arguments allows the reader to not memorize the position of each argument
- Writability (-1 Point)
  - The necessity to explicitly specify keywords of the arguments requires extra effort for the developer.
- Total Point → 2 - 1 = +1
- Extra (+1 Point): If named arguments do not necessarily come after positional arguments so that they are evaluated independent of positional parameters no matter what position they are on. This provides a great flexibility.

No Named Arguments (-1 Points)

Since the reasons and points are the opposite of Parameter Type Checking Criterion, they are not specified to avoid repetition

Default Arguments

Default Arguments (+1 Points)

- Readability (-1 Point)
  - It is harder to understand the relation between the values and parameter variables while each call to the same function may have different call statements.
- Writability (+2 Points)
  - It is much easier to use a default value instead of handling corner cases or null values.
  - Default arguments provide an optional mechanism to write less boilerplate code.
- Total Point → - 1 + 2 = +1

No Default Arguments (-1 Points)

Since the reasons and points are the opposite of Parameter Type Checking Criterion, they are not specified to avoid repetition

Parameter Passing

Pass-by-value for primitive types & pass-by-object-reference for non-primitive types (+3 Points)

- Readability (+1 Point)
  - It is easier to understand the code since there are no other operators for referencing and dereferencing operations.
  - One does not have to know and understand the complicated pointer mechanics when reading the code.
  - It may be a little complicated to understand why some variables are passed with pass-by-reference semantics while others do not as primitive types are passed by value and non-primitive types are passed by object reference
- Writability (+2 Points)
  - It is easier to write the code since there are no other operators for referencing and dereferencing operations.
  - Not interfering with pointers or memory level makes it significantly easier to write a safe code without dangling or null pointers.
  - Once the mechanism is understood it is much easier to write code as there are no different passing methods to complicate the process.
  - One might need to write extra assignment statements to imitate pass-by-reference semantics since the pass-by-value supported semantics reference passing method is not totally pass-by-reference.
- Total Point → 1 + 2 = +3

Pass-by-assignment (pass-by-object-reference) (+4 Points)

- Readability (+2 Points)
  - It is easier to understand the code since there are no other operators for referencing and dereferencing operations.
  - One does not have to know and understand the complicated pointer mechanics when reading the code.
  - One does not have to know about non-primitive and primitive types to differentiate which is passed by value and which is passed by object reference.
  - It may be a little complicated to understand why some assignments persist their side effects while others do not if the reader does not know about mutable and immutable types.
- Writability (+2 Points)
  - It is easier to write the code since there are no other operators for referencing and dereferencing operations.
  - Not interfering with pointers or memory level makes it significantly easier to write a safe code without dangling or null pointers.
  - Once the mechanism is understood it is much easier to write code as there are no different passing methods to complicate the process.
  - One might need to write extra assignment statements to imitate pass-by-reference semantics with pass-by-assignment method since current passing method is not totally pass-by-reference
- Total Point → 2 + 2 = +4

Pass-by-value & Pass-by-reference with explicit pointers (+2 Points)

- Readability (+1 Point)
  - One has to know and understand the complicated pointer mechanics and operations when reading the code.
  - One does not have to know about non-primitive and primitive types to differentiate which is passed by value and which is passed by object reference.
  - It is very clear and certain that when the data is passed-by-reference and when passed-by-value as operations are done at memory level.

- Writability (+1 Point)
  - One does not have to specifically know about types namely which contains the data as value and which contains the address to data as value.
  - One might easily reassign a variable inside the passed function without any extra assignment statements outside the function required.
  - Code writers should be very cautious about dangling pointers to have safe code and constantly write and check referencing and dereferencing operations when writing.

- Total Point → 1 + 1 = +2

Move & Borrow Semantics (-2 Points)

- Readability (-2 Points)
  - One has to know and understand the complicated and unique move and borrow and operations when reading the code.
  - One has to know which types have *Copy* trait so to understand if the variable is passed-by-value or they will be moved.
  - Extra assignment statements to get ownership after the move complicates the code.
  - One does not have to know about non-primitive and primitive types to differentiate which is passed by value and which is passed by object reference.

- Writability (0 Point)
  - Code writers do not have to be very cautious while writing the code as move & borrow semantics guarantee safe code.
  - One might frequently need to write extra assignment statements to get ownership of a variable back after move operations which causes extra effort.

- Total Point → -2 + 0 = -2

Subprogram Passing

Supported with Deep Binding (+3 Points)

- Readability (+2 Points)
  - It is easier to understand the code with deep binding since it has the accustomed static (lexical) scoping semantics.
  - It provides different functionalities to be used within the same function
- Writability (+1 Points)
  - Prevented repetition and decision statements improves writability

- Total Point → 2 + 1 = +3

Supported without Deep Binding (+1 Points)

- Readability (0 Point)
  - It is harder to understand the code without deep binding since it has the accustomed static (lexical) scoping semantics.
  - It provides different functionalities to be used within the same function
- Writability (+1 Points)
  - Prevented repetition and decision statements improves writability

- Total Point → 0 + 1 = +1

2.1. Dart Subprogram Parameters

- Parameter Type Checking (+1 Points)
- Named Arguments (+1 Points + 1 Extra Point)
- Default Arguments (+1 Points)
- Pass-by-value for primitive types & pass-by-object-reference for non-primitive types (+3 Points)
- Supported with Deep Binding (+3 Points)

2.2. Go Subprogram Parameters

- Parameter Type Checking (+1 Points)
- No Named Arguments (-1 Points)
- No Default Arguments (-1 Points)
- Pass-by-value & Pass-by-reference with explicit pointers (+2 Points)
- Supported with Deep Binding (+3 Points)

2.3. JavaScript Subprogram Parameters

- No Parameter Type Checking (-1 Points)
- No Named Arguments (-1 Points)
- Default Arguments (+1 Points)
- Pass-by-value for primitive types & pass-by-object-reference for non-primitive types (+3 Points)
- Supported with Deep Binding (+3 Points)

2.4. Lua Subprogram Parameters

- No Parameter Type Checking (-1 Points)
- No Named Arguments (-1 Points)
- No Default Arguments (-1 Points)
- Pass-by-value for primitive types & pass-by-object-reference for non-primitive types (+3 Points)
- Supported with Deep Binding (+3 Points)

2.5. Python Subprogram Parameters

- No Parameter Type Checking (-1 Points)
- Named Arguments (+1 Points)
- Default Arguments (+1 Points)
- Pass-by-assignment (pass-by-object-reference) (+4 Points)
- Supported with Deep Binding (+3 Points)

2.6. Ruby Subprogram Parameters

- No Parameter Type Checking (-1 Points)
- Named Arguments (+1 Points)
- Default Arguments (+1 Points)
- Pass-by-assignment (pass-by-object-reference) (+4 Points)
- Supported without Deep Binding (+1 Points)

2.7. Rust Subprogram Parameters

- Parameter Type Checking (+1 Points)
- No Named Arguments (-1 Points)
- No Default Arguments (-1 Points)
- Move & Borrow Semantics (-2 Points)
- Supported with Deep Binding (+3 Points)

2.8. Language Evaluation Conclusion and Table

Evaluation Criteria / Languages	Dart	Go	JS	Lua	Python	Ruby	Rust
Type Checking	1	1	-1	-1	-1	-1	1
Keyword (Named) Arguments	2	-1	-1	-1	1	1	-1
Default Arguments	1	-1	1	-1	1	1	-1
Parameter Passing	3	2	3	3	4	4	-2
Subprogram Passing	3	3	3	3	3	1	3

Total Points	10	4	5	3	8	6	0

According to the evaluation, **Dart** programming language possesses the greatest points among the other programming languages evaluated in terms of readability and writability of subprogram parameter operations. Therefore **Dart** seems to be the best programming language when it comes to subprogram parameters.

3. Learning Strategy (Part C)

3.1. Sources Utilized in Common for All Languages

- https://stackoverflow.com/
  - Used for many use case specific problems encountered
  - Many answers are very helpful which can not be found in language documentations which are not beginner friendly
- https://replit.com/
  - Online Compiler For:
    - Dart
    - Go
    - Lua
    - Python
    - Ruby
    - Rust

3.2. Sources Utilized for Dart

- https://dart.dev/guides/language/effective-dart/documentation
  - Used for language reference
- https://www.tutorialspoint.com/dart_programming/index.htm
  - Used for case specific beginner friendly tutorial

3.3. Sources Utilized for Go

- https://go.dev/doc/
  - Used for language reference
- https://www.w3schools.com/go/
  - Used for case specific beginner friendly tutorial

3.4. Sources Utilized for Javascript

- https://javascript.info/
  - Used for language reference

- https://jsfiddle.net/
    - Compiler for JavaScript

3.5. Sources Utilized for Lua

- https://www.lua.org/manual/5.1/manual.html
    - Used for language reference
- https://devdocs.io/lua/
    - Used for language reference
- https://www.tutorialspoint.com/lua/index.htm
    - Used for case specific beginner friendly tutorial

3.6. Sources Utilized for Python

- https://docs.python.org/3/reference/index.html
    - Used for language reference
- https://www.w3schools.com/python/
    - Used for case specific beginner friendly tutorial

3.7. Sources Utilized for Ruby

- https://devdocs.io/ruby~3.2/array
    - Used for language reference
- https://www.tutorialspoint.com/ruby/index.htm
    - Used for case specific beginner friendly tutorial

3.8. Sources Utilized for Rust

- https://doc.rust-lang.org/beta/std/index.html
    - Used for language reference
- https://devdocs.io/rust/
    - Used for language reference
- https://www.tutorialspoint.com/rust/index.htm
    - Used for case specific beginner friendly tutorial

All of the list operation concepts are learned from the sources above and they are used in practice in online compilers replit and JSFiddle. When a struggle or an issue is encountered, I searched for the specific problem or question at stackoverflow.