CS 315

Programming Languages

Homework 02 Report

2022-2023 Spring

Emre Karataş
22001641
Section 01

Instructor: H. Altay Güvenir
Teaching Assistant: Dilruba Sultan Haliloğlu

Table of Contents

A)Design Issues of the Languages:

1. Dart

In Dart language, a function can contain an unlimited number of mandatory positional parameters. After these, the function can include either named parameters or optional positional parameters, but it cannot include both of these parameter types.

Note that below operations are tested on DartPad online editor.

1.1. Are Formal and Actual Parameters Type Checked?

Dart is a statically typed language, checking types at compile-time. It supports type inference allowing developers to omit explicit type declarations. Starting from Dart 2.0, it introduced strong mode which enforces rigorous type checking, providing better safety.

Code Snippet:

```
// this function basically prints the name provided in the argument
void printFamousCSPeople(String name) //formal parameter passed
{
  print(name); // actual parameter passed
}

void main()
{
  printFamousCSPeople("Alan Turing");

}
```

Explanation:

This code snippet declares a function printFamousCSPeople(..) to print the name provided as a parameter. It demonstrates the static type checking in Dart, the passed argument must be a String(*).

(*)As mentioned before, Dart also supports dynamic actual and formal parameters. For example, below code may run without any errors:

```
void printFamousCSPeople(name) //formal parameter passed, param without type
{
  print(name); // actual parameter passed
}

void main()
{
  printFamousCSPeople("Alan Turing");
```

}
However, if one also tries to call printFamousCSpeople(315) in the main function, compiler will raise an error like that:

```
Error compiling to JavaScript:
Info: Compiling with sound null safety.
lib/main.dart:9:3:
Error: Method not found: 'printFamousCSpeople'.
  printFamousCSpeople(315);
  ^^^^^^^^^^^^^^^^^^^
Error: Compilation failed.
```

Erroneous code example is shown in the example code, which is:

```
// ERRONEOUS CODE EXAMPLE
  /*
  printFamousCSPeopleName(315); --- Function expects string parameter
  */
```

Output:

```
Alan Turing
```

1.2. Are Keyword (named) Parameters Supported?

Yes, Dart supports named parameters. One can define them using curly braces {}.

Code Snippet:

```
//Other functions defined in the code not related to this part…

//this function prints the function parameters which are named ones
void printFamousCSPeopleNamedParameters({String? name, int? birthYear})

{
  print('Name: $name, Birth Year: $birthYear');
}

void main()
{

  // other statements in main function not related to this part


 // "Name: Lars Bak, Birth Year: 1965"
  printFamousCSPeopleNamedParameters(name: 'Lars Bak', birthYear: 1965);

  // other statements in main function not related to this part

}
```

Explanation:

This code snippet written in Dart language declares a function named printFamousCSPeopleNamedParameters. The function takes two named parameters, name and birthYear. The parameters are named parameters because they are enclosed in curly braces {}. The ? symbol after the types String and int means that these parameters are nullable, i.e., they can have null values.

If the '?' symbols are not declared, compiler may give an error like that:
The parameter 'name' can't have a value of 'null' because of its type, but the implicit default value is 'null'.

Named parameters are optional unless one include **required** keyword. Note that nullable operator ? is not necessary when the one provides a default value. However, when no default value is provided, one should either mark it as *nullable* or use ***required*** keyword to enforce that a value is passed for that parameter.

Output:

```
Name: Lars Bak, Birth Year: 1965
```

1.3. Are Default Paremeters Supported?

Yes, Dart supports default parameters. One can provide default values for both positional and named parameters. Below code snippet uses named parameters to explain default parameter passing in Dart language.

Code Snippet:

```
//Other functions defined in the code not related to this part…

//this function prints the function parameters which are default
Void printFamousCSPeopleDefaultParameters({String name = 'Tim Berners-Lee', int
birthYear = 1955})
{
   print('Name: $name, Birth Year: $birthYear');
}

void main()
{
  // other statements in main function not related to this part

  // "Name = Tim Berners-Lee, Birth Year = 1955"
  printFamousCSPeopleDefaultParameters();

  // "Name = Tim Berners-Lee, Birth Year = 1922"
  printFamousCSPeopleDefaultParameters(birthYear: 1922);

  // "Name = Barbara Liskov, Birth Year = 1955"
  printFamousCSPeopleDefaultParameters(name: 'Barbara Liskov');
  // "Name = Barbara Liskov, Birth Year = 1939"
  printFamousCSPeopleDefaultParameters(name: 'Barbara Liskov', birthYear: 1939);

  // other statements in main function not related to this part

}
```

Explanation:
This Dart program demonstrates the use of default parameters. The function named printFamousCSPeopleDefaultParameters, takes two named parameters, name and birthYear. Each of these parameters has a default value — if no value is provided when the function is called, these default values will be used. The default value for name is 'Tim Berners-Lee' and for birthYear is 1955(*).

(*)The above code snippet default values are provided for parameters. If the one mistakenly passes a non-string value for name or a non-integer value for birthYear, a compile-time error will occur due to type mismatch.

Erroneous code example is shown in the code, as follows:

```
// ERRONEOUS CODE EXAMPLE
  /*
  printFamousCSPeopleDefaultParameters(name: 2023, birthYear: 1939);
Name must be in String type!
  */
```

Error message:

```
The argument type 'int' can't be assigned to the parameter type 'String'.
```

Output:

```
Name: Tim Berners-Lee, Birth Year: 1955
Name: Tim Berners-Lee, Birth Year: 1922
Name: Barbara Liskov, Birth Year: 1955
Name: Barbara Liskov, Birth Year: 1939
```

1.4. What Are the Paremeter Passing Methods Provided?

In Dart, the parameter passing methods include pass-by-value and pass-by-reference. Primitive types (like int, String, bool) are passed by value, while objects are passed by reference. Dart uses a unified object model approach where everything is an object, even simple types, but they are passed by value. However, when an object is passed, the reference to the object is copied (so it's still a pass-by-value), but the referred object is not; therefore it behaves like pass-by-reference.

Code Snippet:

```
//Other functions defined in the code not related to this part…

// this function changes birthyear and an element of the list provided
```

```
void changeFamousCSPeopleBirthYear(int birthYear, List<String> csPeople)
{
  birthYear = 1965;
  csPeople[0] = "Bjarne Stroustrup"; // changing element in the list
}

void main()
{
  // other statements in main function not related to this part

  int birthYear = 1912;
  List<String> csPeople = ["Alan Turing", "Tim Berners-Lee"];

  changeFamousCSPeopleBirthYear(birthYear, csPeople);

  print('Birth Year: $birthYear'); // prints 1912
  print('CS People: $csPeople'); // prints ["Bjarne Stroustrup", "Tim Berners-Lee"]

  // other statements in main function not related to this part
}
```

Explanation:
The birthYear is a primitive type and it's passed by value, so the change inside changeFamousCSPeopleBirthYear function does not affect the original birthYear value. However, csPeople is a List object, and it's passed by reference. Therefore, changing the list inside the function affects the original list(*).

(*)If the one tries to call this function with a list of other types (like List<int>), one could face a compile-time error. If the one also tries to change an element at an index that does not exist (greater than list length - 1), a runtime error will occur due to index out of range.

```
// ERRONEOUS CODE EXAMPLE 1
 /*
  List<int> bilkentCSJuniorCourses = [315,319,342,353];
  int bilkentBirthYear = 1984;
  changeFamousCSPeopleBirthYear(bilkentBirthYear,bilkentCSJuniorCourses);
 */

// ERRONEOUS CODE EXAMPLE 2 -  ( not shown in the test code, runtime error)

 void changeFamousCSPeopleBirthYear(int birthYear, List<String> csPeople)
{
 birthYear = 1965;
 csPeople[99] = "Bjarne Stroustrup"; // changing element in the list(out of range)
}
```

Error message (for compile time error, erroneous code example 1):
The argument type 'List<int>' can't be assigned to the parameter type 'List<String>'.

Output:

```
Birth Year: 1912
CS People: [Bjarne Stroustrup, Tim Berners-Lee]
```

1.5 Can Subprograms be Passed as Parameters? If so, How is the Referencing Environment of the Passed Subprogram Bound?

In Dart, functions are first-class objects. This means that they can be stored in variables, passed as arguments to other functions, and returned as values from other functions.

Code Snippet:

```
//Other functions defined in the code not related to this part…

//this function takes name and birthyear and prints them.
void printFamousCSPeopleNameAndBirthYear(String name, int birthYear)
{
  print('Name: $name, Birth Year: $birthYear');
}

/*this function calls another function(the one above) inside and prints it in a
decorative way */
void printFamousCSPeopleWithIntroduction(Function printFunction, String name, int
birthYear)
{
  print('------Famous CS Person Details------');
  printFunction(name, birthYear);
  print('------------------------------------');
}

void main()
{
  // other statements in main function not related to this part

  printFamousCSPeopleWithIntroduction(printFamousCSPeopleNameAndBirthYear,
'Alan Turing', 1912);

// other statements in main function not related to this part
}
```

Explanation:

In this Dart program, the function printFamousCSPeopleNameAndBirthYear prints a famous computer scientist's name and birth year when called. One can define another function, printFamousCSPeopleWithIntroduction, that accepts another function as its argument along with a name and birth year. Inside printFamousCSPeopleWithIntroduction, it first prints an introductory line, then calls the passed function with the provided name and birth year, and finally prints a closing line.

In the main function, we call printFamousCSPeopleWithIntroduction and pass printFamousCSPeopleNameAndBirthYear as its argument, along with 'Alan Turing' and 1912 as the name and birth year(*).

(*) If the one mistakenly pass a function that does not have the correct signature (String, int) -> void, a compile-time error will occur. Also, if the programmer passes a function that expects more parameters than he/she supplies when calling it within printFamousCSPeopleWithIntroduction, a runtime error will occur due to missing arguments.

```
// ERRONEOUS FUNCTION TO USE IN MAIN
void incorrectFunction(String name, int birthYear, String nationality)
{
  print('Name: $name, Birth Year: $birthYear, Nationality: $nationality');
}

// ERRONEOUS CODE EXAMPLE (IN MAIN)
  /* this will cause a runtime error because incorrectFunction expects 3 arguments,
  but we're only supplying 2 when calling it within printFamousCSPeopleWithIntroduction
  printFamousCSPeopleWithIntroduction(incorrectFunction, 'Alan Turing', 1912);
  */
```

Error Message:

```
TypeError: A.main__incorrectFunction$closure().call$2 is not a functionError: TypeError:
A.main__incorrectFunction$closure().call$2 is not a function
```

Output:

```
------Famous CS Person Details------
Name: Alan Turing, Birth Year: 1912
----------------------------------------------
```

2. Go

In Go language, A function is a self-contained portion of code that associates input parameters with output parameters. Functions, also referred to as procedures or subroutines, are commonly depicted as a black box, symbolizing their encapsulated nature.

Note that below operations are tested on rextester online editor, which is provided on course website.

2.1. Are Formal and Actual Parameters Type Checked?

Yes, both formal and actual parameters are type checked in Go. Go language is a statically typed one, meaning that the type of a variable is known at compile time.

Code Snippet:

```
// this function basically prints the name provided in the argument
func printFamousCSPeopleName(name string) { //formal parameter passed
	fmt.Println(name) // actual parameter passed
}

func main()
{
	printFamousCSPeopleName("Alan Turing")
}
```

Explanation:
This code snippet basically takes String formal parameter in the Go language and prints it by using actual parameter. In the main function, this subprogram is called by String value “Alan Turing”(*).

(*)One should take care that Go language checks formal and actual parameters type, because the compiler will give an error. For example, as shown in the test code:

```
// ERRONEOUS CODE EXAMPLE
/*
printFamousCSPeopleName(315) --- Function expects string parameter
*/
```

Error Message:

```
cannot use 315 (type int) as type string in argument to printFamousCSPeopleName
```

Output:

```
Alan Turing
```

2.2. Are Keyword (Named) Parameters Supported?

No, Go does not support named parameters or keyword arguments. In Go, arguments must be passed in the same order as the parameters of the function.

Code Snippet:

```
//this function prints the function parameters. Go DOES NOT support Named params.
func printFamousCSPeopleNamedParameters(name string, birthYear int)
{
	fmt.Printf("Name: %s, Birth Year: %d\n", name, birthYear)
}

func main()
{
	printFamousCSPeopleNamedParameters("Lars Bak", 1965)

}
```

Explanation:

This code snippet written in Go language just prints the arguments passed to itself by their positions. Named parameters are not supported in Go language, that means one could not call a function with indicating its parameter names and values(*).

```
(*) // ERRONEOUS CODE EXAMPLE

        /*

   printFamousCSPeopleNamedParameters(name="Lars Bak", birthYear=1965) --- Named parameters are not supported

        */
```

One could not use named parameters in Go.

However, one can also try to achieve effects of named parameters in Go language by using Struct type. *Programmers can define a struct type that represents the parameters for the function, and then pass an instance of that struct to the function. This allows callers to only specify the fields they're interested in and use zero values for the rest.*

```
// Define a struct type to hold the parameters for the function.

type FamousCSPerson struct

 {

        Name string

}

// This function prints the name provided in the argument

func printFamousCSPeopleName(params FamousCSPerson) {

        fmt.Println(params.Name)

}

func main()

{

        // You can now call the function with named parameters, using a struct.

        printFamousCSPeopleName(FamousCSPerson{Name: "Alan Turing"})

}
```

Output of simulating named parameters of the test code can be seen below. (**).

Output:

```
Name: Lars Bak, Birth Year: 1965
Named Parameter Simulation: Ada Lovelace
```

2.3. Are Default Paremeters Supported?

No, Go does not support default parameters. If a function requires certain parameters, the one must provide those parameters when calling the function. To simulate default parameters, one common idiom is to use a variadic function and check if the optional parameter is given.

Code Snippet:

```
//this function simulates default value passing in Go, which actually does not exist.
func printFamousCSPeopleDefaultParameters(name string, birthYear int) {
	if name == "" {
		name = "Tim Berners-Lee"
	}
	if birthYear == 0 {
		birthYear = 1955
	}
	fmt.Printf("Name: %s, Birth Year: %d\n", name, birthYear)
}

func main()
{
	printFamousCSPeopleDefaultParameters("", 0)
	printFamousCSPeopleDefaultParameters("", 1922)
	printFamousCSPeopleDefaultParameters("Barbara Liskov", 0)
	printFamousCSPeopleDefaultParameters("Barbara Liskov", 1939)
}
```

Explanation:

This Go code snippet simulates the behavior of default parameters by using if statements to check whether the arguments passed into the printFamousCSPeopleDefaultParameters function are their zero values, and if so, assigning them default values.

The printFamousCSPeopleDefaultParameters function takes two parameters, name and birthYear. It first checks if name is an empty string (""), which is the zero value for strings in Go. If name is an empty string, it assigns name the default value "Tim Berners-Lee". It does a similar check for birthYear, where it checks if birthYear is 0 (the zero value for integers in Go) and assigns birthYear the default value 1955 if it is. It then prints out the name and birthYear(*).

(*)Noting that Go is statically typed language, one could get error of type mismatching. This problem explained in the section 2.1 in detail.

If the one intended to pass 0 as birthYear or an empty string as name to the function (assuming that these values are meaningful, even though it does not mean a lot in the context of FamousCSPeople example), this function would mistakenly replace them with the default values.

```
// ERRONEOUS CODE EXAMPLE
/*
printFamousCSPeopleDefaultParameters(2023, 1939) --- Function expects string parameter for name
```

```
*/

// ERRONEOUS CODE EXAMPLE
/* The following call will print "Name: Tim Berners-Lee, Birth Year: 1955",
even though the programmer might have intended it to print "Name: , Birth Year: 0"
printFamousCSPeopleDefaultParameters("", 0)
*/
```

Output:

```
Name: Tim Berners-Lee, Birth Year: 1955
Name: Tim Berners-Lee, Birth Year: 1922
Name: Barbara Liskov, Birth Year: 1955
Name: Barbara Liskov, Birth Year: 1939
```

2.4. What Are The Paremeter Passing Methods Provided?

In Go, all parameters are passed by value, which means that the function receives a copy of the original data. If the data is a pointer, then the function receives a copy of the pointer, which can be used to modify the original data.

Code Snippet:

```
// This function tries to change the value of the argument.
func changeBirthYear(birthYear int) {
        birthYear = 1965
}

// this function changes birthyear and an element of the list provided.
func changeFamousCSPeopleBirthYear(birthYear *int, csPeople []string) {
        *birthYear = 1965 // uses the copy of the pointer to change data
        csPeople[0] = "Bjarne Stroustrup" // changing element in the list
}

func main()
{
        birthYear := 1912
        csPeople := []string{"Alan Turing", "Tim Berners-Lee"}

        fmt.Printf("Birth Year before function call: %d\n", birthYear) // Prints 1912
        changeBirthYear(birthYear)
        fmt.Printf("Birth Year after function call: %d\n", birthYear) // Still prints 1912

        fmt.Println("Changing birth year by passing reference...")
        changeFamousCSPeopleBirthYear(&birthYear, csPeople)

        // prints 1965 as change is reflected because of pointer
        fmt.Printf("Birth Year: %d\n", birthYear)

        // prints ["Bjarne Stroustrup", "Tim Berners-Lee"]
        fmt.Printf("CS People: %v\n", csPeople
}
```

Explanation:
In this particular code snippet, passing birthyear as value in changeBirthYear(…) function does not change original data since it is pass by value. However, the function changeFamousCSPeopleBirthYear(…) modifies the original variable by passing the memory address of the variable (a pointer to the variable) as shown in &birthYear. Inside changeFamousCSPeopleBirthYear, dereferencing the pointer with *birthYear to access the original value and change it. That's why the change to birthYear is reflected outside the function(*).

This code snippet also uses slide semantics to change the original data of the array, csPeople.

(*)Usage of pointers is powerful but also tricky in programming languages, one should reference and dereference the pointers carefully not to face potential errors.

Also, if csPeople array is empty, potential errors may occur since csPeople[0] does not exist in that situation.

Also type mismatching is one of potential errors, mentioned above earlier.

```
// ERRONEOUS CODE EXAMPLE
/*
	bilkentCSJuniorCourses := []int{315,319,342,353}
	bilkentBirthYear := 1984
	changeFamousCSPeopleBirthYear(&bilkentBirthYear,bilkentCSJuniorCourses) --- Function expects slice of strings as second parameter
*/
```

Output:
```
Birth Year before function call: 1912
Birth Year after function call: 1912
Changing birth year by passing reference...
Birth Year: 1965
CS People: [Bjarne Stroustrup Tim Berners-Lee]
```

2.5. Can Subprograms Be Passed As Parameters? If So, How Is The Referencing Environment Of The Passed Subprogram Bound?

Yes, Go supports higher-order functions, which means that functions can be passed as parameters to other functions. When a function is passed as an argument, it's called a callback. The referencing environment of the passed subprogram (the function passed as an argument) is bound to the scope where it is defined, not where it is executed. This behavior is known as closure.

Code Snippet:
```
//this function takes name and birthyear and prints them.
func printFamousCSPeopleNameAndBirthYear(name string, birthYear int) {
	fmt.Printf("Name: %s, Birth Year: %d\n", name, birthYear)
}
```

```
//this function calls the function above inside and prints it in a decorative way
func printFamousCSPeopleWithIntroduction(printFunction func(string, int), name
string, birthYear int) {
        fmt.Println("------Famous CS Person Details------")
        printFunction(name, birthYear)
        fmt.Println("------------------------------------")
}

/*createFamousCSIntro creates and returns a new function that uses the captured
variable "name"*/
func createFamousCSIntro(name string) func(int) {
   return func(birthYear int) {
      fmt.Printf("Name: %s, Birth Year: %d\n", name, birthYear)
   }
}

func main()
{
        printFamousCSPeopleWithIntroduction(printFamousCSPeopleNameAndBirthY
ear, "Alan Turing", 1912)

        // Create a new function for each famous CS person
           barbaraLiskov := createFamousCSIntro("Barbara Liskov")
           larryPage := createFamousCSIntro("Larry Page")

           // Call each function with the appropriate birth year
           barbaraLiskov(1939)
           larryPage(1973)

}
```

Explanation:
This Go code demonstrates the use of functions as parameters (also known as first-class functions), as well as simple string and integer manipulation.

The printFamousCSPeopleNameAndBirthYear function takes a string and an int as parameters and prints them formatted as the name and birth year of a famous computer science person.

CreateFamousCSIntro is an example of how to use closure in Go language.

Output:
```
------Famous CS Person Details------
Name: Alan Turing, Birth Year: 1912
------------------------------------
Name: Barbara Liskov, Birth Year: 1939
Name: Larry Page, Birth Year: 1973
```

3. Javascript

Similar to other languages, In Javascript functions may take zero or more parameters to execute the functions.

Note that below operations are tested on OneCompiler online editor and Firefox local machine browser (by selecting inspect button to see output).

3.1. Are Formal And Actual Parameters Type Checked?

JavaScript is a dynamically typed language. This means that the JavaScript interpreter does not check the types of values, and any value can be used with any type of parameter. It is up to the programmer to ensure that the provided values are of the appropriate types, and unexpected behavior can occur if the types are not checked manually by the programmer, even though it is not an error, that is just an incorrect usage.

Code Snippet:

```
function printFamousCSPeopleName(name) { // expects a string
   console.log(name);
}

// correct usage
printFamousCSPeopleName("Alan Turing");

// incorrect usage, but JavaScript doesn't complain
printFamousCSPeopleName(1912);
```

Explanation:
This code snippet basically declares a function to print the name of the famousCSPeople. Even though logically one could expect a name (which is a string), programmer may also provide an integer to execute that function and the compiler will not give any error message(*).

(*) Logical errors may lead to potential bigger errors in the code. As an computer engineering rule, one should define and use the parameters and the functions according to their needs and their logical positions in the program.

Output:

```
Alan Turing
1912
```

3.2. Are Keyword (Named) Parameters Supported?

JavaScript does not support named parameters in the same way that some other languages do (for example, Dart), but programmers can achieve similar functionality by passing an object as a parameter.

Code Snippet:

```
function printFamousCSPeopleDetails({ name, birthYear }) {
   console.log(`Name: ${name}, Birth Year: ${birthYear}`);
}

printFamousCSPeopleDetails({ name: 'Alan Turing', birthYear: 1912 });
```

Explanation:
This JavaScript code snippet defines a function called printFamousCSPeopleDetails which accepts a single parameter - an object with properties name and birthYear. Inside the function, these properties are destructured and then logged to the console.

The function is then called with an object { name: 'Alan Turing', birthYear: 1912 } as the argument, printing "Name: Alan Turing, Birth Year: 1912" to the console(*).

(*)However, there may be possible errors may occur due to this code is Javascript:

Missing properties: If the object passed to printFamousCSPeopleDetails doesn't have name or birthYear properties, undefined will be printed in their place. For example, printFamousCSPeopleDetails({ name: 'Alan Turing' }) would print "Name: Alan Turing, Birth Year: undefined" to the console. Such code example is:

```
printFamousCSPeopleDetails({ name: 'Alan Turing' });  // "Name: Alan Turing, Birth Year: undefined"
```

Incorrect argument type: If the function is called with something other than an object, JavaScript will throw a TypeError. For instance, calling printFamousCSPeopleDetails(1912) would throw an error because the compiler can't destructure properties from a number. Such code example is:

```
// TypeError: Cannot destructure property 'name' of '1912' as it is undefined.

printFamousCSPeopleDetails(1912);
```

Also, calling the function with no parameter and with null parameter will also raise typeError. Code examples are:

```
// TypeError: Cannot destructure property 'name' of 'undefined' as it is undefined.

printFamousCSPeopleDetails();

// TypeError: Cannot destructure property 'name' of 'null' as it is null.

printFamousCSPeopleDetails(null);
```

Output:

```
Name: Alan Turing, Birth Year: 1912
```

3.3. Are Default Paremeters Supported?

Yes, JavaScript supports default parameters. If a parameter is not provided when the function is called, the default value will be used instead.

Code Snippet:

```
function printFamousCSPeopleDetails({ name = "Tim Berners-Lee", birthYear = 1955
} = {}) {
   console.log(`Name: ${name}, Birth Year: ${birthYear}`);
}

printFamousCSPeopleDetails(); // uses defaults
printFamousCSPeopleDetails({ name: 'Alan Turing' }); // uses default birth year
```

Explanation:

This JavaScript code defines a function called printFamousCSPeopleDetails that takes a single parameter - an object with properties name and birthYear. These properties are destructured from the input object and each property is assigned a default value.

The function uses JavaScript's default parameters feature, where the object to be destructured defaults to an empty object {} if no argument is provided, and the properties name and birthYear default to "Tim Berners-Lee" and 1955 respectively, if those properties are not present in the object.

(*)If the programmer call the function with an argument that's not an object, JavaScript will throw a TypeError because JS can't destructure properties from a non-object. For example, calling printFamousCSPeopleDetails(1912) would result in an error.

Output:

```
Name: Tim Berners-Lee, Birth Year: 1955
Name: Alan Turing, Birth Year: 1955
```

3.4. What Are The Paremeter Passing Methods Provided?

JavaScript provides pass-by-value for primitive data types (number, string, boolean, null, undefined, and symbol) and pass-by-reference for objects (including arrays and functions).

Code Snippet:

```
// Parameter Passing Methods Example
   function changeDetails(name, person) {
      name = "Bjarne Stroustrup";
      person.name = "Bjarne Stroustrup";
   }

   let name = "Alan Turing";
   let person = { name: "Alan Turing" };

   changeDetails(name, person);
```

```
    console.log(name); // still "Alan Turing"
    console.log(person.name); // changed to "Bjarne Stroustrup"
```

Explanation:
This JavaScript code snippet demonstrates the two parameter passing methods in JavaScript as mentioned above: pass-by-value and pass-by-reference.

When changeDetails(name, person) is called, name is passed by value (because it's a primitive type), and person is passed by reference (because it's an object). So when the subprogram modify name inside, it doesn't affect the name variable outside the function. But when the programmer modifies the person object inside the function, the changes are reflected outside the function because the reference points to the same object(*).

(*)Passing null, undefined or not passing an array to reflect change outside of function may lead unintended results and/or errors.

Output:

```
Alan Turing
Bjarne Stroustrup
```

3.5. Can Subprograms Be Passed As Parameters? If So, How Is The Referencing Environment Of The Passed Subprogram Bound?

Yes, functions in JavaScript are first-class objects and can be passed as parameters to other functions. The referencing environment of the passed subprogram is bound to the scope in which it was defined, not where it was executed.

Code Snippet:

```
// Subprograms as Parameters Example
    function printFamousCSPeopleWithIntroduction(printFunction, name, birthYear) {
        console.log('------Famous CS Person Details------');
        printFunction(name, birthYear);
        console.log('-----------------------------------');
    }

    function printFamousCSPeopleNameAndBirthYear(name, birthYear) {
        console.log(`Name: ${name}, Birth Year: ${birthYear}`);
    }

    printFamousCSPeopleWithIntroduction(printFamousCSPeopleNameAndBirthYear,
"Alan Turing", 1912);
```

Explanation:
The function printFamousCSPeopleWithIntroduction is declared, which takes three parameters: a function printFunction, and two data parameters name and birthYear. Inside printFamousCSPeopleWithIntroduction, two lines are logged to the console, with a call to printFunction(name, birthYear) in between. This means it's using the

passed function with the given parameters name and birthYear.The function printFamousCSPeopleNameAndBirthYear is declared, which takes name and birthYear as parameters and logs them to the console.Finally, printFamousCSPeopleWithIntroduction is called, passing printFamousCSPeopleNameAndBirthYear as the printFunction parameter and "Alan Turing" and 1912 as the name and birthYear parameters, respectively(*).

(*)If the programmer passes something other than a function for printFunction, trying to call it as a function will throw a TypeError.

(*) If the programmer passes a function that expects a different number of parameters for printFunction, it may lead to unexpected behavior. JavaScript doesn't enforce the number of parameters a function should receive, but if a function expects more parameters than it receives, the missing parameters will be undefined.

Output:

```
------Famous CS Person Details------

Name: Alan Turing, Birth Year: 1912

-----------------------------------
```

4. Lua

In Lua, an argument is like a placeholder. When a function is invoked, one could pass a value to the argument. This value is referred to as the actual parameter or argument. The parameter list refers to the type, order, and number of the arguments of a method. Arguments are optional; that is, a method may contain no argument.

Note that below operations are tested on rextester online editor, which is provided on course website.

4.1. Are Formal And Actual Parameters Type Checked?

Lua is dynamically typed, meaning that variables don't have a predefined type, they instead get their type based on the value they are holding at a particular moment. So, there is no compile-time type checking for parameters. However, the one can perform manual type checking using the type() function.

Code Snippet:

```
function printFamousCSPerson(name, birthYear)
   if type(name) ~= 'string' or type(birthYear) ~= 'number' then
      error('Invalid parameters')
   end
   print('Name: '..name..', Birth Year: '..birthYear)
end

printFamousCSPerson('Alan Turing', 1912)
```

Explanation:

In this particular code snippet, the function printFamousCSPerson checks if name is a string and if birthYear is a number(*).

(*)If they're not in the type of their intended values, it throws an error. The main error that could occur here is if the programmers pass parameters of a different type than expected.

Output:

```
Name: Alan Turing, Birth Year: 1912
```

4.2. Are Keyword (Named) Parameters Supported?

Lua doesn't directly support named parameters in the way some other languages do. However, programmers can pass a table to a function, and Lua tables can have named fields, effectively giving the programmers the effects of named parameters.

Code Snippet:

```
-- Named parameters
function printFamousCSPersonNamed(details)
   print('Name: '..details.name..', Birth Year: '..details.birthYear)
end

printFamousCSPersonNamed({name = 'Barbara Liskov', birthYear = 1939})
```

Explanation:

Here in this code snippet , a table is used to pass named parameters to the function printFamousCSPerson(*).

(*) The main error that could occur here is if the programmer forget to pass the table or if the table doesn't contain the expected fields.

Output:

```
Name: Barbara Liskov, Birth Year: 1939
```

4.3. Are Default Paremeters Supported?

Lua doesn't directly support default parameters, but programmers can achieve similar behavior using the or operator.

Code Snippet:

```
function printFamousCSPersonDefault(name, birthYear)
   name = name or 'Elon Musk'
   birthYear = birthYear or 1923
   print('Name: '..name..', Birth Year: '..birthYear)
end

printFamousCSPersonDefault()
```

Explanation:
In this particular code snippet, the or operator is used to set default values for name and birthYear if they are not provided.

Output:
```
Name: Elon Musk, Birth Year: 1923
```

4.4. What Are The Paremeter Passing Methods Provided?

Lua uses pass-by-value semantics. However, tables in Lua are actually references to data, so when programmers pass a table to a function, they are passing a reference to that table, effectively allowing them to modify the original table.

Code Snippet:
```
-- Parameter passing methods
function changeFamousCSPerson(person)
   person.name = 'Changed Name'
end


famousPerson = {name = 'Ada Lovelace'}
changeFamousCSPerson(famousPerson)
print(famousPerson.name)  -- prints 'Changed Name'
```

Explanation:
In this example, the table famousPerson is passed to the function changeFamousCSPerson, and the change to person.name inside the function affects famousPerson outside the function.

Output:
```
Changed Name
```

4.5. Can Subprograms Be Passed As Parameters? If So, How Is The Referencing Environment Of The Passed Subprogram Bound?

Lua supports first-class functions as other languages mentioned above, meaning functions can be passed as parameters to other functions. The environment of the passed subprogram is bound to the scope where it's defined, not where it's executed.

Code Snippet:
```
-- Subprograms as parameters
function printWithFunction(func, name, birthYear)
   print('***')
   func(name, birthYear)
   print('***')
end

function printFamousCSPersonSub(name, birthYear)
   print('Name: '..name..', Birth Year: '..birthYear)
```

```
end

printWithFunction(printFamousCSPersonSub, 'Larry Page', 1973)
```

Explanation:
In this example code snippet, the function printFamousCSPerson is passed to printWithFunction, which then calls printFamousCSPerson(*).

(*)The potential error here could be passing something other than a function to printWithFunction.

Output:
```
***
Name: Larry Page, Birth Year: 1973
***
```

5. Python

A function is a block of code which only runs when it is called. One can pass data, known as parameters, into a function. A function can return data as a result.

Note that below operations are tested on rextester online editor (by Python3), which is provided on course website.

5.1. Are Formal And Actual Parameters Type Checked?

Python is dynamically typed, meaning that types are checked at runtime rather than at compile-time. Python 3.5 introduced optional type hints, which can be used to suggest the expected types of function parameters and return values.

Code Snippet:
```
def print_famous_cs_people_details(name: str, birth_year: int):
    print(f'Name: {name}, Birth Year: {birth_year}')

print_famous_cs_people_details("Alan Turing", 1912)
```

Explanation:
This code snippet basically prints the name and birthyear. Python does not care about parameters type. However, with Python 3.5 the type hints (: str and : int) are purely informational and do not enforce type checking. If the programmer call this function with arguments of the wrong type, Python will not raise a compile-time error.

Output:
```
Name: Alan Turing, Birth Year: 1912
```

5.2. Are Keyword (Named) Parameters Supported?

Yes, Python supports named parameters. Programmers can specify parameters by their names regardless of their order in the function definition.

Code Snippet:
```
# Named Parameters Example
def print_famous_cs_people_details(name, birth_year):
    print(f'Name: {name}, Birth Year: {birth_year}')

print_famous_cs_people_details(birth_year=1939, name="Barbara Liskov")
```

Explanation:

This code snippet shows that one could use named parameters in Python and one can also give the parameters regardless of their order.

Output:
```
Name: Barbara Liskov, Birth Year: 1939
```

5.3. Are Default Paremeters Supported?

Yes, Python supports default parameters. Programmers can specify default values for parameters that will be used if no argument is provided for the parameter when the function is called.

Code Snippet:
```
# Default Parameters Example
def print_famous_cs_people_details(name="Ada Lovelace", birth_year=1815):
    print(f'Name: {name}, Birth Year: {birth_year}')

print_famous_cs_people_details() # uses the default parameters
print_famous_cs_people_details(birth_year=1945)
print_famous_cs_people_details(name="John Doe")
```

Explanation:

This code snippet shows how to implement default parameter passing in Python functions. Function declaration itself provides name and birth_year, which can be called without any parameter since default parameters will be returned if there will be no specified value in the function call.

Output:
```
Name: Ada Lovelace, Birth Year: 1815
Name: Ada Lovelace, Birth Year: 1945
Name: John Doe, Birth Year: 1815
```

5.4. What Are The Paremeter Passing Methods Provided?

Python uses a mechanism known as "Call by Object Reference" or "Call by Sharing". This means that each formal parameter of the function gets a copy of each reference in the argument. In the case of mutable objects (like lists or dictionaries), if programmers modify the object, the change will be reflected outside the function. But if the programmers rebind the reference inside the function, the change won't propagate outside.

Code Snippet:
```
# Parameter Passing Methods Example
def change_famous_cs_people_details(name, details):
    name = "Bjarne Stroustrup"
    details['name'] = "Bjarne Stroustrup"

name = "Elon Musk"
details = {'name': "Jeff Bezos"}

change_famous_cs_people_details(name, details)

print(name)  # prints "Elon Musk"
print(details['name'])  # prints "Bjarne Stroustrup"
```

Explanation:
This code snippet basically implements how to pass parameters in Python and their corresponding outputs. Change_famous_cs_people function changes array type element and this change reflects outside of the function. However, passing primitive type does not change the data outside of the function.

Output:
```
Elon Musk
Bjarne Stroustrup
```

5.5. Can Subprograms Be Passed As Parameters? If So, How Is The Referencing Environment Of The Passed Subprogram Bound?

Yes, in Python, functions are first-class objects. Programmers can pass them around as arguments. When programmers pass a function as an argument, the referencing environment of the passed function is the scope where it was defined, which is known as lexical scoping.

Code Snippet:
```
Subprograms as Parameters Example
def print_famous_cs_people_details(name, birth_year):
    print(f'Name: {name}, Birth Year: {birth_year}')

def print_with_introduction(func, name, birth_year):
    print('------Famous CS Person Details------')
    func(name, birth_year)
    print('------------------------------------')

print_with_introduction(print_famous_cs_people_details, "Larry Page", 1973)
```

Explanation:
This code snippet basically shows how to pass a function to another function as a parameter in Python. Arguments to be used to passed function should also be passed to function, to execute passed function inside the main function(*).

(*)If a programmer defines a function inside another function, and the inner function references a variable from the outer function, the inner function will retain a reference to that variable even after the outer function has finished executing. This can lead to unexpected behavior if the programmer is not familiar with the concept of closures.

Output:
```
------Famous CS Person Details------
Name: Larry Page, Birth Year: 1973
------------------------------------
```

6. Ruby

Parameters in Ruby programming language are variables that are defined in method definition, and which represent the ability of a method to accept arguments.

Note that below operations are tested on rextester online editor, which is provided on course website.

6.1. Are Formal And Actual Parameters Type Checked?

Ruby is a dynamically typed language, meaning that variable types are checked during runtime and not at compile time. As such, formal and actual parameters in Ruby are not explicitly type checked.

Code Snippet:
```
# Type Checking Example
def print_famous_cs_people (name, birth_year)
  puts "Name: #{name}, Birth Year: #{birth_year}"
end

print_famous_cs_people("Alan Turing", 1912)

# This also works but the output is not as expected.
print_famous_cs_people(1912, "Alan Turing")
```

Explanation:
This code snippet shows that Ruby actually does not check types of the formal or actual parameters, which can lead potential logical errors in the program(*).

(*)An error in this context might be passing parameters in an order or type that the function isn't expecting.

Output:
```
Name: Alan Turing, Birth Year: 1912
Name: 1912, Birth Year: Alan Turing
```

6.2. Are Keyword (Named) Parameters Supported?

Yes, Ruby supports named parameters. They provide more readability to the code written.

Code Snippet:

```
# Named Parameters Example
def print_famous_cs_people_named_params(name:, birth_year:)
  puts "Name: #{name}, Birth Year: #{birth_year}"
end

print_famous_cs_people_named_params(birth_year: 1939, name: "Barbara Liskov")
```

Explanation:

Named parameters are very useful in Ruby as in other languages. User may call the function with any order they want. For example, in this code, the positions of name and birth year are changed during calling the function(*).

(*)If you try to call the function without the required named parameters, it will raise an ArgumentError.

Output:

```
Name: Barbara Liskov, Birth Year: 1939
```

6.3. Are Default Paremeters Supported?

Yes, Ruby supports default parameters. If a parameter is not provided when the function is called, Ruby will use the default value.

Code Snippet:

```
# Default Parameters Example
def print_famous_cs_people_default_params(name = "Ada Lovelace", birth_year =
1815)
  puts "Name: #{name}, Birth Year: #{birth_year}"
end

print_famous_cs_people_default_params()

def print_famous_cs_people_default_params_2(birth_year: 1815, name: "Ada
Lovelace")
  puts "Name: #{name}, Birth Year: #{birth_year}"
end

print_famous_cs_people_default_params_2(name: "John Doe")
print_famous_cs_people_default_params_2(birth_year:2000)
```

Explanation:

This code shows how to use default parameters when the function is called without explicit parameters passed to the function. “Ada Lovelace” and 1815 are default, and will be printed out even the programmer does not specify them in the calling function.

If someone provides any parameter different than default ones, compiler will take into account that parameters.

However, it's not possible to supply the second without also supplying the first. Instead of using positional parameters, one should try keyword parameters.

This code shows(also written above) shows how to achieve that:

```
def print_famous_cs_people_default_params_2(birth_year: 1815, name: "Ada
Lovelace")
  puts "Name: #{name}, Birth Year: #{birth_year}"
end

print_famous_cs_people_default_params_2(name: "John Doe")
print_famous_cs_people_default_params_2(birth_year:2000)
```

Output:

```
Name: Ada Lovelace, Birth Year: 1815
Name: John Doe, Birth Year: 1815
Name: Ada Lovelace, Birth Year: 2000
```

6.4. What Are The Paremeter Passing Methods Provided?

Ruby supports "pass by value" but due to its dynamic nature, it can seem like "pass by reference" when dealing with mutable objects. In reality, it's passing the reference by value.

Code Snippet:

```
# Parameter Passing Methods Example
def change_famous_cs_people_param_passing(name, details)
  name = "Bjarne Stroustrup"
  details[:name] = "Bjarne Stroustrup"
end

name = "Elon Musk"
details = {name: "Jeff Bezos"}

change_famous_cs_people_param_passing(name, details)

puts name  # prints "Elon Musk"
puts details[:name]  # prints "Bjarne Stroustrup"
```

Explanation:

This Ruby code shows the difference between pass-by-value and pass-by-reference.

The change_famous_cs_people_param_passing method is designed to change the values of the variables name and details. In Ruby, Strings are mutable but they are passed by value, not by reference. So, the variable name remains "Elon Musk" even after the method call because the change is made to a copy, not the original name variable.

On the other hand, the details variable is a hash which is a complex data type. These are passed by reference in Ruby. Thus, when we change details inside the method, the original details hash is updated to {name: "Bjarne Stroustrup"}(*).

(*)The potential errors can occur if the programmer tries to change details without it being a hash or not providing the correct key for change. Also, one should always remember to use the : before name when trying to access the value from the details hash, i.e., details[:name], otherwise it will result in nil because name is treated as a variable and not a symbol.

Output:
```
Elon Musk
Bjarne Stroustrup
```

6.5. Can Subprograms Be Passed As Parameters? If So, How Is The Referencing Environment Of The Passed Subprogram Bound?

Yes, Ruby supports passing methods (subprograms) as parameters using Procs and lambdas. The referencing environment of the passed subprogram is bound lexically in Ruby. This is often referred to as a closure. When a proc or lambda (which are both types of closures in Ruby) is created, it remembers the environment in which it was defined. This includes any variables that were in scope at the time of definition. Therefore, it has access to these variables even when it is called in a different scope.

Code Snippet:
```
# Subprograms as Parameters Example
def print_famous_cs_people_sub(name, birth_year)
  puts "Name: #{name}, Birth Year: #{birth_year}"
end

def print_with_introduction(func, name, birth_year)
  puts '------Famous CS Person Details------'
  func.call(name, birth_year)
  puts '------------------------------------'
end

print_func = Proc.new do |name, birth_year|
  print_famous_cs_people_sub(name, birth_year)
end

print_with_introduction(print_func, "Larry Page", 1973)
```

Explanation:
In this code, print_name_and_birth_year is a Proc (a type of closure), which is passed to the print_famous_cs_people_with_introduction method. Inside this

method, the print_function proc is invoked with the call method, passing in the name and birth_year parameters. If there were any variables in scope at the time print_name_and_birth_year was defined, it would have access to those even within the print_famous_cs_people_with_introduction method due to the lexical scoping of the closure(*).

(*)Potential errors might include failing to use the call method to invoke the proc, or providing the wrong number or type of arguments to either the method or the proc.

Output:
```
------Famous CS Person Details------
Name: Larry Page, Birth Year: 1973
-----------------------------------
```

7. Rust

In Rust language, one can define functions to have parameters, which are special variables that are part of a function's signature. When a function has parameters, the one can provide it with concrete values for those parameters.

Note that below operations are tested on rextester online editor, which is provided on course website.

7.1. Are Formal And Actual Parameters Type Checked?

Rust is a statically typed language, which means that it must know the types of all variables at compile time. The Rust compiler will not automatically convert types. To convert one type to another, one should need to explicitly use the as keyword. This applies to the function parameters as well. If one tries to pass parameters of a different type than specified in the function definition, Rust will throw a compile-time error.

Code Snippet:
```
fn print_details(name: &str, birth_year: i32) {
        println!("Name: {}, Birth Year: {}", name, birth_year);
   }

   print_details("Alan Turing", 1912); // Correct
   //print_details(1912, "Alan Turing"); This would cause a compile time error
```

Explanation:
This code snippet written in Rust are waiting parameters in the type they defined since Rust is a statically typed language. This function basically prints the name and birth year. Passing birth year before name throws compile time error since it is not defined in that way in the function declaration.

Output:
Name: Alan Turing, Birth Year: 1912

7.2. Are Keyword (Named) Parameters Supported?

Named parameters are not supported in Rust. This means when one calls a function, one must provide the parameters in the same order they were defined in the function signature.

However, there are some workarounds for simulating named parameters in Rust, mostly revolving around the use of structs.

Code Snippet:
```
// Named Parameters Example: Rust does not support named parameters
   //However, one could simulate named parameters
   struct FamousCSParams {
   name: String,
   birth_year: i32,
   }

   fn simulate_named_params(params: FamousCSParams) {
      println!("Birth Year: {}, Name: {}", params.birth_year, params.name);
   }

   simulate_named_params(FamousCSParams {
      name: "Jeff Bezos".to_string(),
      birth_year: 1964,
   });
```

Explanation:
This code snippet in Rust illustrates a way to simulate named parameters using a struct.A new struct type, FamousCSParams, is declared with two fields, name of type String, and birth_year of type i32. A function simulate_named_params is declared that takes an argument params of type FamousCSParams. Inside this function, we print the birth_year and name fields of the params struct.The function simulate_named_params is called with an instance of the FamousCSParams struct. The struct instance is created inline with the name field set to "Jeff Bezos", and birth_year set to 1964.

By using this approach, one can effectively simulate named parameters: programmers have to specify the names of the parameters (i.e., the struct's fields) when creating the struct instance.

Output:
Birth Year: 1964, Name: Jeff Bezos

7.3. Are Default Paremeters Supported?

Rust does not support default parameters. All function arguments in Rust are required unless you use an Option type which allows for None as a valid value.

However, there are some workarounds for simulating default parameters in Rust, mostly revolving around the use of structs.

Code Snippet:
```
// Default Parameters Example: Rust does not support default parameters
   // one could simulate default parameters in Rust by struct type.
   #[derive(Default)]
    struct FamousCSParams2 {
   name: String,
   birth_year: i32,
   }
   impl FamousCSParams2 {
   fn new() -> Self {
       Self {
           name: "Ada Lovelace".to_string(),
           birth_year: 1815,
           ..Default::default()
           }
       }
   }

   fn simulate_default_params(params: FamousCSParams2) {
   println!("Name: {}, Birth Year: {}", params.name, params.birth_year);
   }

   simulate_default_params(FamousCSParams2::new());
```

Explanation:
Here in this code snippet, programmer used Rust's Default trait to assign default values to the struct's fields. The new method then allows programmer to easily generate a new FamousCSParams struct with these default values.

Output:
```
Name: Ada Lovelace, Birth Year: 1815
```

7.4. What Are The Paremeter Passing Methods Provided?

When a function gets called in Rust, the values of the actual parameters get copied, or more technically correct, they get moved to the function's formal parameters. After the move, the actual parameters can no longer be accessed. However, Rust supports passing by reference, meaning that instead of copying the value, a reference to the existing value is used.

Code Snippet:
```
fn change_famous_cs_people_value(mut name: String, mut birth_year: i32) {
   name = "Bjarne Stroustrup".to_string();
   birth_year = 1950;
   // changes to name and birth_year are local to this function
}

fn change_famous_cs_people_reference(name: &mut String, birth_year: &mut i32) {
    *name = "Bjarne Stroustrup".to_string();
    *birth_year = 1950;
    // changes to name and birth_year will affect the original variables
}

let mut name = "Elon Musk".to_string();
let mut birth_year = 1971;

// pass by value
change_famous_cs_people_value(name.clone(), birth_year);
// Prints "Name: Elon Musk, Birth Year: 1971"
println!("Name: {}, Birth Year: {}", name, birth_year);

// pass by reference
change_famous_cs_people_reference(&mut name, &mut birth_year);
// Prints "Name: Bjarne Stroustrup, Birth Year: 1950"
println!("Name: {}, Birth Year: {}", name, birth_year);
```

Explanation:
The change_famous_cs_people_value function takes its parameters by value. This means that name and birth_year are cloned, and the function works with these copies. Any changes it makes will not affect the originals.

The change_famous_cs_people_reference function, on the other hand, takes mutable references to name and birth_year (&mut String, &mut i32). In this case, the function doesn't get its own copy of name and birth_year - it directly modifies the originals via the provided references.

As a result, changes made by change_famous_cs_people_value don't reflect outside its scope, while change_famous_cs_people_reference does change the original values(*).

(*)If the one forgets to use the mut keyword when declaring name and birth_year, one will get a compile-time error as Rust enforces immutability by default. Also, one should be aware of Rust's ownership rules. For example, if the one was to use name instead of name.clone() in change_famous_cs_people_value, Rust would prevent her/him from using name after that point because the function took ownership of name.

Output:
```
Name: Elon Musk, Birth Year: 1971
Name: Bjarne Stroustrup, Birth Year: 1950
```

7.5. Can Subprograms Be Passed As Parameters? If So, How Is The Referencing Environment Of The Passed Subprogram Bound?

In Rust, one can pass functions as parameters. The environment of the passed subprogram (function) is bound to where the function is defined.

Code Snippet:
```
// Subprograms as Parameters Example
   fn print_with_introduction(print_func: fn(&str, i32), name: &str, birth_year: i32) {
      println!("---Famous CS Person---");
      print_func(name, birth_year);
      println!("----------------------");
   }

   print_with_introduction(print_details, "Larry Page", 1973);
```

Explanation:
This Rust code is demonstrating how functions can be passed as parameters to other functions - a concept also known as "Higher Order Functions". In the example, print_with_introduction is a function that takes three parameters: another function print_func of type fn(&str, i32), a string slice name and an integer birth_year. The function print_func itself takes a string slice and an integer as parameters. Within the body of print_with_introduction, the passed function print_func is called with the parameters name and birth_year. This demonstrates that a function can be used as an argument and can be executed within another function. In the last line, the function print_with_introduction is called with print_details(*).

(*)If the types of name and birth_year don't match the ones expected by the print_func (i.e., &str and i32, respectively), the Rust compiler will also generate an error.Any modifications to the passed parameters name and birth_year within print_func won't affect the original values, since they're passed by value (not mutable references). If the intention was to modify these, it would lead to a logical error.

Output:
```
---Famous CS Person---
Name: Larry Page, Birth Year: 1973
----------------------
```

B) Evaluations of These Languages in Terms of Readability and Writability:

1. Dart

1.1. Readability:

Dart language provides good readability in terms of subprogram parameters. The use of type annotations for formal parameters makes it clear what types of arguments are expected. This helps developers understand the function signature and ensure the correct types of arguments are passed. Additionally, named parameters enhance readability by providing descriptive names for arguments, making it easier to understand their purpose. Default parameters also contribute to readability by reducing the need for repetitive or unnecessary arguments when calling functions.

1.2. Writability:

Dart language offers decent writability for subprogram parameters. The support for named parameters allows developers to specify arguments by name, making the code more expressive and flexible. This helps in changing the order of arguments without breaking existing function calls. Default parameters improve writability by reducing the need to provide values for every parameter, making function calls more concise and less error-prone. Additionally, Dart's pass-by-value parameter passing method simplifies code writing as it avoids unexpected side effects on the original data.

2. Go

2.1. Readability:

In terms of readability, go language provides a clear and concise syntax for defining and calling subprogram parameters. The function definitions clearly specify the types of formal parameters, making it easy to understand the expected input. When calling functions, the arguments are passed in a straightforward manner, allowing for easy comprehension of the parameter values being provided. The code snippets given above demonstrate the readability of Go language, where the function names and parameter names are descriptive, and the syntax is clean and readable.

2.2 Writability:

Go supports various parameter passing methods such as pass by value and pass by reference. It allows for passing parameters by reference using pointers, enabling functions to modify the original values of variables. This provides flexibility and control over data manipulation within functions. Additionally, Go's support for multiple return values enhances writability by allowing functions to return multiple values simultaneously. The code snippets in the provided example showcase the writability of Go language, demonstrating the ability to define functions, pass parameters, and modify values effectively.

3. Javascript

3.1. Readability:

The JavaScript language provides readable subprogram parameter handling. In the code snippets provided above, the function signatures clearly indicate the expected parameters, and the function calls provide explicit values for those parameters. This makes it easy to understand the purpose and intent of each function call.

The use of destructuring in the printFamousCSPeopleDetails function allows for named parameters, improving readability by providing clear labels for each parameter. Similarly, the printFamousCSPeopleDetailsDefault function demonstrates the use of default parameters, which further enhances readability by specifying fallback values if no arguments are provided.

The subprogram passing example printFamousCSPeopleWithIntroduction demonstrates the readability of passing functions as parameters. The function name and purpose clearly convey the intention of the code, making it easy to understand the behavior being implemented.

3.2. Writability:

The language allows flexibility in defining functions with different parameter types and does not enforce strict type checking on actual parameters. This flexibility allows developers to write code quickly and experiment with different parameter configurations.

The support for named parameters and default parameters in JavaScript enhances writability by providing options for expressing intent and handling optional values. This allows developers to define functions with meaningful names and provide default values for parameters, reducing the need for excessive conditional logic.

The ability to pass subprograms as parameters further enhances writability by enabling the implementation of higher-order functions and function composition. This allows developers to write concise and reusable code by abstracting common functionality into separate subprograms.

4. Lua

4.1. Readability:

In Lua, the readability of subprogram parameters can vary depending on the approach taken. Lua does not have built-in type checking for formal and actual parameters, so it requires manual type checking within the subprogram. This can make the code less readable and more error-prone, as demonstrated in the printFamousCSPersonWithTypeCheck function. The type checking adds complexity to the code and makes it harder to understand at a glance.

The use of named parameters in Lua, as shown in the printFamousCSPersonNamed function, can improve readability. By passing a table with named fields as a single argument, it becomes clear what each parameter represents. This makes the code more self-documenting and easier to understand.

The concept of default parameters is also present in Lua, as demonstrated in the printFamousCSPersonDefault function. While it provides flexibility, it can also make the code less readable, especially if there are multiple parameters with default values. The meaning of the parameters and their default values may not be immediately apparent to someone reading the code.

4.2. Writability:

Lua provides good writability when it comes to subprogram parameters. It has a flexible syntax that allows for different parameter passing methods. However, as mentioned before, the lack of built-in type checking and default parameter support can impact writability to some extent. Developers need to manually handle type checking and set default values, which requires more code and effort.

The ability to pass subprograms as parameters, demonstrated in the printWithFunction function, enhances writability. It allows for code reuse and promotes modular programming. Developers can pass different functions to perform specific tasks, making the code more adaptable and reusable.

Overall, Lua provides decent writability for subprogram parameters, but the absence of built-in type checking and default parameters can make the code more verbose and potentially error-prone. However, the flexibility and support for passing subprograms as parameters contribute to its writability by enabling code reuse and modularity.

5. Python

5.1. Readability:

In the given Python test code given above, subprogram parameters are defined clearly and succinctly, and the nature of these parameters is communicated in a straightforward manner, making the code easy to read and understand.

Type Checking: Python's type hinting (name: str, birth_year: int) is optional and does not affect program execution but serves as documentation and tooling, making the code more readable as it clearly states the expected types of the parameters.

Named Parameters: Python's support for named parameters, as shown in the test code aboce, allows for clearer calls to functions. When calling print_famous_cs_people_details, using named parameters makes it clear which argument corresponds to which parameter, improving readability.

Default Parameters: Python's default parameters allow the program to work even if certain parameters aren't provided in the function call. This feature simplifies function calls while still maintaining the functionality of the function, leading to more readable code.

Parameter Passing: Python's simplicity in parameter passing (by reference for mutable objects and by value for immutable ones) also helps enhance readability. The mechanism is intuitive and doesn't confuse readers with complicated passing schemes.

Subprograms as Parameters: Python's capability to pass subprograms as parameters is demonstrated without introducing much complexity. This supports creating higher order functions, making it easier to build complex functionality from simpler functions.

5.2. Writability:

Type Checking: While Python is dynamically typed, it allows optional type hinting, which can aid in writing more robust code by making the expected types of values explicit.

Named Parameters: Python's support for named parameters enhances the writability, as it provides flexibility in the order of arguments while calling a function. This flexibility can be helpful when a function has many parameters or when it is desirable to make the role of an argument in a function call clear.

Default Parameters: Default parameters make Python functions more flexible and easier to write. A function can be written with sensible defaults, which can then be overridden when necessary.

Parameter Passing: Python's way of passing parameters, whether mutable or not, is simple and straightforward. This simplicity makes it easy to write functions without having to worry about the intricacies of parameter passing.

Subprograms as Parameters: Being able to pass subprograms as parameters to other subprograms, as shown in the example, is a powerful feature. It leads to higher levels of abstraction and more modular code, facilitating code reuse, which improves writability.

6. Ruby

6.1. Readability:

Ruby's design principles heavily favor readability, and this is evident in how it handles subprogram parameters. The language allows for clear definition and usage of functions and methods. This includes being able to provide default values for parameters, using named parameters for clarity, and being able to pass methods or functions as parameters.

Type Checking: Ruby's dynamic typing makes it easy to call a function with arguments of any type. While this can lead to unexpected behavior if used incorrectly (as shown in the example code above where a birth year is provided instead of a name), it also allows for greater flexibility.

Named Parameters: Named parameters in Ruby add a great deal of readability to the code. Instead of relying on the order of the arguments, you can use the names to make it clear what each argument is for.

Default Parameters: Default parameters also improve readability by allowing a function to be called with fewer arguments without having to explicitly pass nil or a default value each time.

Parameter Passing Methods: The example of changing the name within a function demonstrates how Ruby treats strings as mutable, but the changes do not persist outside the function due to Ruby's pass-by-value semantics. This is an important feature to understand when reading Ruby code, as it affects how functions can alter their arguments.

Subprograms as Parameters: Being able to pass a function as a parameter is another feature that can greatly enhance readability. It allows for more complex and modular code, where functions can be built up from smaller, reusable parts.

6.2. Writability:

Ruby's handling of subprogram parameters also contributes to its writability. Its flexibility, combined with the use of clear, human-readable syntax, makes it easy to write complex functions.

Type Checking: The lack of strict type checking in function parameters makes it easier to write functions that can handle different types of input.

Named Parameters: Named parameters allow programmers to call a function without having to remember the exact order of parameters, making it easier to write correct calls to functions.

Default Parameters: Having default parameters makes it simpler to write function calls, as programmers do not always need to provide all the parameters if defaults are set. This makes writing code faster and less error-prone.

Parameter Passing Methods: The way Ruby handles parameter passing allows programmers to modify the parameters within the function. This can simplify the code inside the function.

Subprograms as Parameters: This feature is not used as frequently as the others, but it can greatly simplify the code when used correctly. It allows for higher-level functions that can take other functions as parameters, enabling a higher level of abstraction. This can lead to more reusable and modular code.

7. Rust

7.1. Readability:

Rust's strict typing system and requirement for explicit handling of variables and functions enhances readability in many aspects. With strong static typing, Rust ensures that the types of parameters for subprograms are clear and understood, reducing ambiguity and the chance of runtime errors.

However, Rust does not directly support named parameters and default parameters. The workaround using structs (as shown in this code snippets above) can be a bit more verbose and might reduce readability for programmers who are not used to this idiom, especially compared to languages which directly support these features.

When passing parameters by reference, Rust makes it very clear by using the &mut keyword, leaving no room for confusion about whether the original variable can be modified within the function or not. This improves readability as it's clear to see which parameters might be changed inside the function.

The ability to pass subprograms as parameters is also clear in Rust, thanks to its strong typing system. It's straightforward to see what the function's signature must be, enhancing readability.

7.2. Writability:

Rust's strong typing system means that the programmer must be very explicit about types when writing code, which can sometimes make writing Rust code more time-consuming compared to dynamically-typed languages.

The lack of direct support for named parameters and default parameters also impacts writability, as using structs for this purpose involves more code and more complexity. This is a disadvantage compared to languages which directly support these features.

Passing parameters by reference in Rust requires use of the &mut keyword and a clear understanding of Rust's ownership and borrowing system. This means more effort is required when writing code, but also less chance of unexpected bugs.

While being able to pass subprograms as parameters increases Rust's expressiveness and flexibility, understanding the syntax and type signatures can be more challenging compared to some other languages. However, once understood, it is a powerful feature that can help create clean, reusable code.

8. General Evaluation:

I believe that Python has the best structure to work with in terms of readability and writability of the code in the given context of subprogram parameters. It's syntax rules are clear. Compared to other languages, the workload of the programmer is less since Python provides all functionalities that are needed for this homework evaluation, all of them are easier to read and write, which are the most important aspects of a language in terms of evaluation.

C) Learning Strategy:

My learning strategy for this homework is to test and show with their examples about the required design issues listed in the homework instructions with the potential errors which may occur due to these designs. I generally used my main sources for the languages that I listed below. Besides that, I also check regarding Stackoverflow posts to see and to determine how to act these possible errors. I also tried to make the examples simple enough to understand concepts better. Same concept and functionalities are used through the homework to see differantations between languages and their implementations.

Here are the main resources that I used during my research phase:

For Dart, I used the documentation which can be found at Dart website: https://dart.dev/language/functions

For Go (Golang), I used I used popular W3 schools tutorial website: https://www.w3schools.com/go/go_function_parameters.php

For Javascript, I used popular W3 schools tutorial website: https://www.w3schools.com/js/js_function_parameters.asp

For Lua, I used Luascript.dev website regarding function parameters: https://www.luascript.dev/blog/function-parameters

For Python I used popular W3 schools tutorial website regarding Python function arguments: https://www.w3schools.com/python/gloss_python_function_arguments.asp

For Ruby, I checked this online website rubyguides to learn how to implement ruby method arguments: https://www.rubyguides.com/2018/06/rubys-method-arguments/

For Rust, I used formal documentation about functions inside the Rust developer website: https://doc.rust-lang.org/reference/items/functions.html

The online compilers that I used during testing phase:

Dart: https://dartpad.dev/?

GO, Lua, Python, Ruby, Rust: https://rextester.com/

Javascript (as HTML): https://onecompiler.com/html

(One may use Firefox or Google Chrome for JS and may inspect the code result from the console.)