CS 315 - Spring 2023

Homework - 1

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

1. Operations in Each Language (Part A)

1.1. Dart

1.1.1. Code Sample

```dart
void main() {
  var operationNumber = 1;

  // Operation 1: Declare/create an empty list
  print("\nOperation " + operationNumber.toString());
  var languages = [];
  printResultingList(languages, operationNumber);
  operationNumber = operationNumber + 1;

  // Operation 2: Initialize a list with some values
  print("\nOperation " + operationNumber.toString());
  languages = ["Dart", "Go", "Javascript", "Lua", "Python", "Ruby", "Rust"];
  printResultingList(languages, operationNumber);
  operationNumber = operationNumber + 1;

  // Operation 3: Check if the list is empty or not
  print("\nOperation " + operationNumber.toString());
  isListEmpty(languages);
  printResultingList(languages, operationNumber);
  operationNumber = operationNumber + 1;

  // Operation 4: Add a new element to a list
  print("\nOperation " + operationNumber.toString());
  var newElement = "SQL";
  languages.add(newElement);
  printResultingList(languages, operationNumber);
  operationNumber = operationNumber + 1;

  // Operation 5: Check if a particular element exists in the list
  print("\nOperation " + operationNumber.toString());
  doesContain(languages, newElement);
  printResultingList(languages, operationNumber);
  operationNumber = operationNumber + 1;

  // Operation 6: Remove a particular element from the list
  print("\nOperation " + operationNumber.toString());
  languages.remove(newElement);
```

```
  printResultingList(languages, operationNumber);
  operationNumber = operationNumber + 1;

  // Operation 7: Get the head and the tail of a list
  print("\nOperation " + operationNumber.toString());
  print("Head of the list is: " + languages.first);
  print("Tail of the list is: " + languages.sublist(1).join(" "));
  printResultingList(languages, operationNumber);
  operationNumber = operationNumber + 1;

  // Operation 8: Print all of the elements in the list
  print("\nOperation " + operationNumber.toString());
  printResultingList(languages, operationNumber);
  operationNumber = operationNumber + 1;
}

void isListEmpty(var list) {
  if (list.isEmpty) {
    print("List is empty");
  } else {
    print("List is not empty");
  }
}

void doesContain(var list, var object) {
  if (list.contains(object)) {
    print("List contains " + object);
  } else {
    print("List does not contain " + object);
  }
}

void printResultingList(var list, var operationNumber) {
  print("List after Operation " +
      operationNumber.toString() +
      ": " +
      list.join(" "));
}
```

1.1.2. Execution Result

```
Operation 1
List after Operation 1:

Operation 2
List after Operation 2: Dart Go Javascript Lua Python Ruby Rust

Operation 3
List is not empty
List after Operation 3: Dart Go Javascript Lua Python Ruby Rust

Operation 4
List after Operation 4: Dart Go Javascript Lua Python Ruby Rust SQL

Operation 5
List contains SQL
List after Operation 5: Dart Go Javascript Lua Python Ruby Rust SQL

Operation 6
List after Operation 6: Dart Go Javascript Lua Python Ruby Rust

Operation 7
Head of the list is: Dart
Tail of the list is: Go Javascript Lua Python Ruby Rust
List after Operation 7: Dart Go Javascript Lua Python Ruby Rust

Operation 8
List after Operation 8: Dart Go Javascript Lua Python Ruby Rust
```

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

An empty growable list is initialized with empty list literal. It can be done also with List.empty() function call, yet the returning list would be ungrowable.

1.1.2.3. Operation 2

The empty list is initialized with a string list literal containing programming language names.

1.1.2.4. Operation 3

The list object is passed into the *isListEmpty* function which accepts a list variable. *isListEmpty* function that I defined uses the built-in *isEmpty* property of lists to check emptiness of a list which has a bool type. Then the corresponding result is printed within an if-else block.

1.1.2.5. Operation 4

A new string object is added to the list with the built-in *add* function belonging to Lists, which accepts an object as a parameter.

1.1.2.6. Operation 5

The list and a string object which holds a value identical to that of an object in the list is passed into the *doesContain* function which accepts a list and an object variable. *doesContain* function that I defined uses the built-in *contains* function which accepts an object formal parameter and returns a bool value after checking if the list includes the object or not. Then the corresponding result is printed within an if-else block.

1.1.2.7. Operation 6

A string object which holds a value identical to that of an object in the list is passed into the built-in *remove* function.

1.1.2.8. Operation 7

Head (first element) of the list is accessed by the built-in *first* property of the list. Tail (sublist excluding the first element) of the list is accessed by the built-in *sublist* function accepting an index parameter. Number 1 is used as a formal parameter to get a sublist of the main list from the second element (index 1) until the last element. Moreover the built-in *join* function of list is used for converting the list object to a string object with proper format (spaces between elements).

1.1.2.9. Operation 8

The list object is passed into the *printResultingList* function which accepts a list and an operationNumber variable. *printResultingList* function that I defined uses the built-in *join* function of list which is used for converting the list object to a string object with proper format (spaces between elements). Then the concatenated string is printed to show the list's final state. Finally, non-string numbers can be converted to string with *toString* function.

1.2. Go

1.2.1. Code Sample

```go
package main

import (
    "fmt"
    "strings"
)

func main() {
    operationNumber := 1

  // Operation 1: Declare/create an empty list
    fmt.Println(fmt.Sprintf("\nOperation %v", operationNumber))
    languages := []string{}
    printResultingList(languages, operationNumber)
    operationNumber = operationNumber + 1

  // Operation 2: Initialize a list with some values
    fmt.Println(fmt.Sprintf("\nOperation %v", operationNumber))
    languages = []string{"Dart", "Go", "Javascript", "Lua", "Python", "Ruby",
"Rust"}
    printResultingList(languages, operationNumber)
    operationNumber = operationNumber + 1

  // Operation 3: Check if the list is empty or not
    fmt.Println(fmt.Sprintf("\nOperation %v", operationNumber))
    isListEmpty(languages)
    printResultingList(languages, operationNumber)
    operationNumber = operationNumber + 1

  // Operation 4: Add a new element to a list
    fmt.Println(fmt.Sprintf("\nOperation %v", operationNumber))
    newElement := "SQL"
    languages = append(languages, newElement)
    printResultingList(languages, operationNumber)
    operationNumber = operationNumber + 1

  // Operation 5: Check if a particular element exists in the list
    fmt.Println(fmt.Sprintf("\nOperation %v", operationNumber))
```

```
    doesContain(languages, newElement)
    printResultingList(languages, operationNumber)
    operationNumber = operationNumber + 1

  // Operation 6: Remove a particular element from the list
    fmt.Println(fmt.Sprintf("\nOperation %v", operationNumber))
    languages = removeFromList(languages, newElement)
    printResultingList(languages, operationNumber)
    operationNumber = operationNumber + 1

  // Operation 7: Get the head and the tail of a list
    fmt.Println(fmt.Sprintf("\nOperation %v", operationNumber))
    first := languages[0]
    last := languages[1:len(languages)]
    fmt.Println("Head of the list is: " + first)
    fmt.Println("Tail of the list is: " + strings.Join(last, " "))
    printResultingList(languages, operationNumber)
    operationNumber = operationNumber + 1

  // Operation 8: Print all of the elements in the list
    fmt.Println(fmt.Sprintf("\nOperation %v", operationNumber))
    printResultingList(languages, operationNumber)
    operationNumber = operationNumber + 1
}

func isListEmpty(list []string) {
    if len(list) > 0 {
        fmt.Println("List is not empty")
    } else {
        fmt.Println("List is empty")
    }
}

func doesContain(list []string, object string) int {
    for i, value := range list {
        if value == object {
            fmt.Println("List contains " + object)
            return i
        }
    }

    fmt.Println("List does not contain " + object)
```

```
    return 0
}

func removeFromList(list []string, object string) []string {
    for index, value := range list {
        if value == object {
            return append(list[:index], list[index+1:]...)
        }
    }
    return list
}

func printResultingList(list []string, operationNumber int) {
    fmt.Println(fmt.Sprintf("List after Operation %v: %s", operationNumber,
strings.Join(list, " ")))
}
```

1.2.2. Execution Result

```
Operation 1
List after Operation 1:

Operation 2
List after Operation 2: Dart Go Javascript Lua Python Ruby Rust

Operation 3
List is not empty
List after Operation 3: Dart Go Javascript Lua Python Ruby Rust

Operation 4
List after Operation 4: Dart Go Javascript Lua Python Ruby Rust SQL

Operation 5
List contains SQL
List after Operation 5: Dart Go Javascript Lua Python Ruby Rust SQL

Operation 6
List after Operation 6: Dart Go Javascript Lua Python Ruby Rust

Operation 7
Head of the list is: Dart
Tail of the list is: Go Javascript Lua Python Ruby Rust
List after Operation 7: Dart Go Javascript Lua Python Ruby Rust
```

```
Operation 8
List after Operation 8: Dart Go Javascript Lua Python Ruby Rust
```

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

An empty slice (a built-in variable for referencing arrays) is initialized with empty slice literal. It can be done also with languages := make([]string, 0) function call, yet the internal call to runtime.makeslice increases the overhead cost.

1.2.3.2. Operation 2

The empty slice is initialized with a string slice literal containing programming language names (Nearly identical to array string literal yet for arrays there is required a number representing array size between the square brackets).

1.2.3.3. Operation 3

The slice object is passed into the *isListEmpty* function which accepts a slice variable. *isListEmpty* function that I defined uses the built-in *len* function which accepts a slice/array object and returns the length of it. The returning value is used for checking the emptiness of a slice. Then the corresponding result is printed within an if-else block.

1.2.3.4. Operation 4

A new string object is added to the slice with the built-in *append* function belonging to slices, which accepts an object as parameter.

1.2.3.5. Operation 5

The slice and a string object which holds a value identical to that of an object in the slice is passed into the *doesContain* function which accepts a slice and an object variable. *doesContain* function that I defined utilizes a for loop to iterate over the slice and checks each time if the passed object equals to the object at the current index through the loop. Then the corresponding result is printed if there is an equality between objects in the loop and function returns. Otherwise not found message is printed.

1.2.3.6. Operation 6

The slice and a string object which holds a value identical to that of an object in the slice is passed into the *removeFromList* function which accepts a slice and an object variable. *removeFromList* function that I defined utilizes a for loop to iterate over the slice and checks each time if the passed object equals the object at the current index through the loop. If there is an equality between the passed object and the object at the current iteration index, then two sub slices before and the after the current index are appended to imitate a remove operation. Otherwise the passed slice is returned.

1.2.3.7. Operation 7

Head (first element) of the slice is accessed with the built-in *subscript operator* of the slice by accessing element at index 0. Tail (subslice excluding the first element) of the slice is accessed with the built-in *subscript slicing operator* of the slice by accessing subslice between the second element (at index 1) and the last element (at index len(slice) which returns length of the slice). Moreover the built-in *Join* function of *strings* is used for converting the slice object to a string object with proper format (spaces between elements).

1.2.3.8. Operation 8

The slice object is passed into the *printResultingList* function which accepts a slice and an operationNumber variable. *printResultingList* function that I defined uses the built-in *Sprintf* function which is used for formatting string objects. Moreover, the built-in *join* function of slice which is used for converting the slice object to a string object with proper format (spaces between elements). Then the concatenated string is printed to show the slice's final state. Finally, non-string numbers can be converted to string with *str* function.

1.3. Javascript

1.3.1. Code Sample

```
<script>

  function main() {
    var operationNumber = 1;

    // Operation 1: Declare/create an empty list
    console.log("\nOperation " + operationNumber);
```

```
    var languages = [];
    printResultingList(languages, operationNumber);
    operationNumber = operationNumber + 1;

    // Operation 2: Initialize a list with some values
    console.log("\nOperation " + operationNumber.toString());
    languages = ["Dart", "Go", "Javascript", "Lua", "Python", "Ruby", "Rust"];
    printResultingList(languages, operationNumber);
    operationNumber = operationNumber + 1;

    // Operation 3: Check if the list is empty or not
    console.log("\nOperation " + operationNumber.toString());
    isListEmpty(languages);
    printResultingList(languages, operationNumber);
    operationNumber = operationNumber + 1;

    // Operation 4: Add a new element to a list
    console.log("\nOperation " + operationNumber.toString());
    var newElement = "SQL";
    languages.push(newElement);
    printResultingList(languages, operationNumber);
    operationNumber = operationNumber + 1;

    // Operation 5: Check if a particular element exists in the list
    console.log("\nOperation " + operationNumber.toString());
    doesContain(languages, newElement);
    printResultingList(languages, operationNumber);
    operationNumber = operationNumber + 1;

    // Operation 6: Remove a particular element from the list
    console.log("\nOperation " + operationNumber.toString());
    languages.splice(languages.indexOf(newElement), 1);
    printResultingList(languages, operationNumber);
    operationNumber = operationNumber + 1;

    // Operation 7: Get the head and the tail of a list
    console.log("\nOperation " + operationNumber.toString());
    console.log("Head of the list is: " + languages[0]);
    console.log("Tail of the list is: " + languages.slice(1,
languages.length).join(" "));
    printResultingList(languages, operationNumber);
    operationNumber = operationNumber + 1;
```

```
  // Operation 8: Print all of the elements in the list
  console.log("\nOperation " + operationNumber.toString());
  printResultingList(languages, operationNumber);
  operationNumber = operationNumber + 1;
}

function isListEmpty(list) {
  if (list.length == 0) {
    console.log("List is empty");
  } else {
    console.log("List is not empty");
  }
}

function doesContain(list, object) {
  if (list.includes(object)) {
    console.log("List contains " + object);
  } else {
    console.log("List does not contain " + object);
  }
}

function printResultingList(list, operationNumber) {
  console.log("List after Operation " +
    operationNumber +
    ": " +
    list.join(" "));
}

main();


</script>
```

1.3.2. Execution Result

```
“
Operation 1"
"List after Operation 1: "
"
Operation 2"
"List after Operation 2: Dart Go Javascript Lua Python Ruby Rust"
"
Operation 3"
"List is not empty"
"List after Operation 3: Dart Go Javascript Lua Python Ruby Rust"
"
Operation 4"
"List after Operation 4: Dart Go Javascript Lua Python Ruby Rust SQL"
"
Operation 5"
"List contains SQL"
"List after Operation 5: Dart Go Javascript Lua Python Ruby Rust SQL"
"
Operation 6"
"List after Operation 6: Dart Go Javascript Lua Python Ruby Rust"
"
Operation 7"
"Head of the list is: Dart"
"Tail of the list is: Go Javascript Lua Python Ruby Rust"
"List after Operation 7: Dart Go Javascript Lua Python Ruby Rust"
"
Operation 8"
"List after Operation 8: Dart Go Javascript Lua Python Ruby Rust"
```

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

An empty array is initialized with an empty array literal.

1.3.3.2. Operation 2

The empty array is initialized with an array literal containing programming language names.

1.3.3.3. 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 *length* property of arrays returning the size of the array. Then the corresponding result is printed within an if-else block depending on the returned size of the array.

1.3.3.4. Operation 4

A new string object is added to the array with the built-in *push* function belonging to arrays, which accepts an object as parameter.

1.3.3.5. Operation 5

The array and a string object which holds a value identical to that of an object in the array is passed into the *doesContain* function which accepts an array and an object variable. *doesContain* function that I defined uses the built-in *includes* function which accepts an object formal parameter and returns a bool value after checking if the array includes the object or not. Then the corresponding result is printed within an if-else block.

1.3.3.6. Operation 6

A string object which holds a value identical to that of an object in the array is passed into the built-in *splice* function which accepts an index value and a number value to remove a given number of values starting from a given index. Moreover built-in *indexOf* function of arrays is used which accepts an object and returns the index of the passed object if it exists in the array.

1.3.3.7. Operation 7

Head (first element) of the array is accessed with the built-in *subscript operator* of the array by accessing element at index 0. Tail (subarray excluding the first element) of the array is accessed with the built-in *slice* function of the arrays which accepts two index values and returns the subarray between passed indexes. Tail is from the second element (index at 1) until the last element (index is found by *length* property of arrays). Moreover the built-in *join* function of *strings* is used for converting the array object to a string object with proper format (spaces between elements).

1.3.3.8. Operation 8

The array object is passed into the *printResultingList* function which accepts an array and an operationNumber variable. *printResultingList* function that I defined which does string

concatenation operations by + operator. Moreover, the built-in *join* function of array which is used for converting the array object to a string object with proper format (spaces between elements). Then the concatenated string is printed to show the array's final state. Finally, non-string numbers can be converted to string with *toString* function.

1.4. Lua

1.4.1. Code Sample

```lua
function main()
  local operationNumber = 1

  -- Operation 1: Declare/create an empty list
  print("\nOperation " .. operationNumber)
  local languages = {}
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1

  -- Operation 2: Initialize a list with some values
  print("\nOperation " .. operationNumber)
  languages = { "Dart", "Go", "Javascript", "Lua", "Python", "Ruby", "Rust" }
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1

  -- Operation 3: Check if the list is empty or not
  print("\nOperation " .. operationNumber)
  isListEmpty(languages)
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1

  -- Operation 4: Add a new element to a list
  print("\nOperation " .. operationNumber)
  local newElement = "SQL"
  table.insert(languages, newElement)
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1

  -- Operation 5: Check if a particular element exists in the list
  print("\nOperation " .. operationNumber)
  doesContain(languages, newElement)
  printResultingList(languages, operationNumber)
```

```
  operationNumber = operationNumber + 1

  -- Operation 6: Remove a particular element from the list
  print("\nOperation " .. operationNumber)
  removeFromList(languages, newElement)
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1

  -- Operation 7: Get the head and the tail of a list
  print("\nOperation " .. operationNumber)
  local first = languages[1]
  local tail = { unpack(languages, 2) }
  print("Head of the list is: " .. first)
  print("Tail of the list is: " .. table.concat(tail, " "))
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1

  -- Operation 8: Print all of the elements in the list
  print("\nOperation " .. operationNumber)
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1
end

function isListEmpty(list)
  if next(list) == nil then
    print("List is empty")
  else
    print("List is not empty")
  end
end

function doesContain(list, object)
  for _, value in pairs(list) do
    if value == object then
      print("List contains " .. object)
      return
    end
  end
  print("List does not contain " .. object)
end

function removeFromList(list, object)
```

```
  for index, value in pairs(list) do
    if value == object then
      table.remove(list, index)
      return
    end
  end
end

function printResultingList(list, operationNumber)
  print("List after Operation " .. operationNumber .. ": " ..
table.concat(list, " "))
end

main()
```

1.4.2. Execution Result

```
Operation 1
List after Operation 1:

Operation 2
List after Operation 2: Dart Go Javascript Lua Python Ruby Rust

Operation 3
List is not empty
List after Operation 3: Dart Go Javascript Lua Python Ruby Rust

Operation 4
List after Operation 4: Dart Go Javascript Lua Python Ruby Rust SQL

Operation 5
List contains SQL
List after Operation 5: Dart Go Javascript Lua Python Ruby Rust SQL

Operation 6
List after Operation 6: Dart Go Javascript Lua Python Ruby Rust

Operation 7
Head of the list is: Dart
Tail of the list is: Go Javascript Lua Python Ruby Rust
List after Operation 7: Dart Go Javascript Lua Python Ruby Rust

Operation 8
List after Operation 8: Dart Go Javascript Lua Python Ruby Rust
```

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

An empty table is initialized with an empty table literal.

1.4.3.2. Operation 2

The empty table is initialized with a string list literal containing programming language names.

1.4.3.3. Operation 3

The table object is passed into the *isListEmpty* function which accepts a table variable. *isListEmpty* function that I defined uses the built-in *next* function which accepts a table object and returns the next index of the said table and its value associated with the table, however when the passed table is empty it returns nil (null). The returning value is used for checking the emptiness of a table. Then the corresponding result is printed within an if-else block.

1.4.3.4. Operation 4

A new string object is added to the table with the built-in *insert* function belonging to tables, which accepts an object as parameter.

1.4.3.5. Operation 5

The table and a string object which holds a value identical to that of an object in the table is passed into the *doesContain* function which accepts a table and an object variable. *doesContain* function that I defined utilizes a for loop to iterate over the table and checks each time if the passed object equals to the object at the current index through the loop. Then the corresponding result is printed if there is an equality between objects in the loop and function returns. Otherwise not found message is printed.

1.4.3.6. Operation 6

The table and a string object which holds a value identical to that of an object in the table is passed into the *removeFromList* function which accepts a table and an object variable. *removeFromList* function that I defined utilizes a for loop to iterate over the table and checks each time if the passed

object equals the object at the current index through the loop. If there is an equality between the passed object and the object at the current iteration index, then the built-in *remove* function which belongs to tables, accepting a list and object variables is utilized. Otherwise the passed table is returned.

1.4.3.7. Operation 7

Head (first element) of the table is accessed with the built-in *subscript operator* of the table by accessing element at index 1. Tail (subtable excluding the first element) of the table is accessed with the built-in *unpack* function which accepts a list object and a starting index value which returns multiple values from the passed index until to end. Moreover the built-in *concat* function of tables is used for converting the table object to a string object with proper format (spaces between elements).

1.4.3.8. Operation 8

The table object is passed into the *printResultingList* function which accepts a table and an operationNumber variable. *printResultingList* function that I defined which does string concatenation operations by .. operator. Moreover, the built-in *concat* function of tables which is used for converting the array object to a string object with proper format (spaces between elements). Then the concatenated string is printed to show the slice's final state.

1.5. Python

1.5.1. Code Sample

```python
def main():
  operationNumber = 1

  # Operation 1: Declare/create an empty list
  print("\nOperation " + str(operationNumber))
  languages = []
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1

  # Operation 2: Initialize a list with some values
  print("\nOperation " + str(operationNumber))
  languages = ["Dart", "Go", "Javascript", "Lua", "Python", "Ruby", "Rust"]
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1
```

```
  # Operation 3: Check if the list is empty or not
  print("\nOperation " + str(operationNumber))
  isListEmpty(languages)
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1

  # Operation 4: Add a new element to a list
  print("\nOperation " + str(operationNumber))
  newElement = "SQL"
  languages.append(newElement)
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1

  # Operation 5: Check if a particular element exists in the list
  print("\nOperation " + str(operationNumber))
  doesContain(languages, newElement)
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1

  # Operation 6: Remove a particular element from the list
  print("\nOperation " + str(operationNumber))
  languages.remove(newElement)
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1

  # Operation 7: Get the head and the tail of a list
  print("\nOperation " + str(operationNumber))
  print("Head of the list is: " + languages[0])
  print("Tail of the list is: " + " ".join(languages[1:]))
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1

  # Operation 8: Print all of the elements in the list
  print("\nOperation " + str(operationNumber))
  printResultingList(languages, operationNumber)
  operationNumber = operationNumber + 1


def isListEmpty(list):
  if list:
    print("List is not empty")
  else:
```

```
    print("List is empty")


def doesContain(list, object):
  if object in list:
    print("List contains " + object)
  else:
    print("List does not contain " + object)


def printResultingList(list, operationNumber):
  print("List after Operation " + str(operationNumber) + ": " + "
".join(list))


main()
```

1.5.2. Execution Result

```
Operation 1
List after Operation 1:

Operation 2
List after Operation 2: Dart Go Javascript Lua Python Ruby Rust

Operation 3
List is not empty
List after Operation 3: Dart Go Javascript Lua Python Ruby Rust

Operation 4
List after Operation 4: Dart Go Javascript Lua Python Ruby Rust SQL

Operation 5
List contains SQL
List after Operation 5: Dart Go Javascript Lua Python Ruby Rust SQL

Operation 6
List after Operation 6: Dart Go Javascript Lua Python Ruby Rust

Operation 7
Head of the list is: Dart
Tail of the list is: Go Javascript Lua Python Ruby Rust
List after Operation 7: Dart Go Javascript Lua Python Ruby Rust
```

```
Operation 8
List after Operation 8: Dart Go Javascript Lua Python Ruby Rust
```

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

An empty list is initialized with empty list literal.

1.5.3.2. Operation 2

The empty list is initialized with a string list literal containing programming language names.

1.5.3.3. Operation 3

The list object is passed into the *isListEmpty* function which accepts a list variable. *isListEmpty* function that I defined uses directly the list value in an if block since in Python empty sequences or sequences with length 0 are considered false. Then the corresponding result is printed within an if-else block.

1.5.3.4. Operation 4

A new string object is added to the list with the built-in *append* function belonging to Lists, which accepts an object as a parameter.

1.5.3.5. Operation 5

The list and a string object which holds a value identical to that of an object in the list is passed into the *doesContain* function which accepts a list and an object variable. *doesContain* function that I defined uses the built-in *in* operator which returns a bool value after checking if the list includes the object or not. Then the corresponding result is printed within an if-else block.

1.5.3.6. Operation 6

A string object which holds a value identical to that of an object in the list is passed into the built-in *remove* function.

1.5.3.7. Operation 7

Head (first element) of the list is accessed with the built-in subscript operator of the list by accessing element at index 0. Tail (sublist excluding the first element) of the list is accessed with the built-in subscript slicing operator of the list by accessing the sublist between the second element (at index 1) and the last element (Implies until the end of the list if left empty). Moreover the built-in *join* function of strings is used for converting the list object to a string object with proper format (spaces between elements).

1.5.3.8. Operation 8

The list object is passed into the *printResultingList* function which accepts a list and an operationNumber variable. *printResultingList* function that I defined which does string concatenation operations by + operator. Moreover, the built-in *join* function of list which is used for converting the list object to a string object with proper format (spaces between elements). Then the concatenated string is printed to show the lists's final state.

1.6. Ruby

1.6.1. Code Sample

```ruby
def main
  operationNumber = 1

  # Operation 1: Declare/create an empty list
  puts "\nOperation " + operationNumber.to_s
  languages = []
  printResultingList(languages, operationNumber)
  operationNumber += 1

  # Operation 2: Initialize a list with some values
  puts "\nOperation " + operationNumber.to_s
  languages = %w[Dart Go Javascript Lua Python Ruby Rust]
  printResultingList(languages, operationNumber)
  operationNumber += 1

  # Operation 3: Check if the list is empty or not
  puts "\nOperation " + operationNumber.to_s
  isListEmpty(languages)
  printResultingList(languages, operationNumber)
```

```
  operationNumber += 1

  # Operation 4: Add a new element to a list
  puts "\nOperation " + operationNumber.to_s
  newElement = 'SQL'
  languages.push(newElement)
  printResultingList(languages, operationNumber)
  operationNumber += 1

  # Operation 5: Check if a particular element exists in the list
  puts "\nOperation " + operationNumber.to_s
  doesContain(languages, newElement)
  printResultingList(languages, operationNumber)
  operationNumber += 1

  # Operation 6: Remove a particular element from the list
  puts "\nOperation " + operationNumber.to_s
  languages.delete(newElement)
  printResultingList(languages, operationNumber)
  operationNumber += 1

  # Operation 7: Get the head and the tail of a list
  puts "\nOperation " + operationNumber.to_s
  puts 'Head of the list is: ' + languages.first
  puts 'Tail of the list is: ' + languages.slice(1, languages.length).join('
')
  printResultingList(languages, operationNumber)
  operationNumber += 1

  # Operation 8: Print all of the elements in the list
  puts "\nOperation " + operationNumber.to_s
  printResultingList(languages, operationNumber)
  operationNumber += 1
end

def isListEmpty(list)
  if list.empty?
    puts 'List is empty'
  else
    puts 'List is not empty'
  end
end
```

```ruby
def doesContain(list, object)
  if list.include?(object)
    puts 'List contains ' + object
  else
    puts 'List does not contain ' + object
  end
end

def printResultingList(list, operationNumber)
  puts 'List after Operation ' +
       operationNumber.to_s +
       ': ' +
       list.join(' ')
end

main
```

1.6.2. Execution Result

```
Operation 1
List after Operation 1:

Operation 2
List after Operation 2: Dart Go Javascript Lua Python Ruby Rust

Operation 3
List is not empty
List after Operation 3: Dart Go Javascript Lua Python Ruby Rust

Operation 4
List after Operation 4: Dart Go Javascript Lua Python Ruby Rust SQL

Operation 5
List contains SQL
List after Operation 5: Dart Go Javascript Lua Python Ruby Rust SQL

Operation 6
List after Operation 6: Dart Go Javascript Lua Python Ruby Rust

Operation 7
Head of the list is: Dart
Tail of the list is: Go Javascript Lua Python Ruby Rust
List after Operation 7: Dart Go Javascript Lua Python Ruby Rust

Operation 8
```

```
List after Operation 8: Dart Go Javascript Lua Python Ruby Rust
```

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

An empty array is initialized with an empty array literal. It can be also done with the Array.new call which is semantically the same.

1.6.3.2. Operation 2

The empty array is initialized with an array literal containing programming language names with %w[elements] structure which uses whitespace as separator. It can also be done with classical array literal separated by commas which semantically the same.

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

A new string object is added to the array with the built-in *push* function belonging to arrays, which accepts an object as parameter.

1.6.3.5. Operation 5

The array and a string object which holds a value identical to that of an object in the array is passed into the *doesContain* function which accepts an array and an object variable. *doesContain* function that I defined uses the built-in *include?* function which accepts an object formal parameter and returns a bool value after checking if the array includes the object or not. Then the corresponding result is printed within an if-else block.

1.6.3.6. Operation 6

A string object which holds a value identical to that of an object in the array is passed into the built-in *remove* function of arrays which accepts an object and removes it from the array function called from.

1.6.3.7. Operation 7

Head (first element) of the array is accessed with the built-in *first* property of the array. Tail (subarray excluding the first element) of the array is accessed with the built-in *slice* function of the arrays which accepts two index values and returns the subarray between passed indexes. Tail is from the second element (index at 1) until the last element (index is found by *length* property of arrays). Moreover the built-in *join* function of *strings* is used for converting the array object to a string object with proper format (spaces between elements).

1.6.3.8. Operation 8

The array object is passed into the *printResultingList* function which accepts an array and an operationNumber variable. *printResultingList* function that I defined which does string concatenation operations by + operator. Moreover, the built-in *join* function of array which is used for converting the array object to a string object with proper format (spaces between elements). Then the concatenated string is printed to show the array's final state. Finally, non-string numbers can be converted to string with *to_s* function.

1.7. Rust

1.7.1. Code Sample

```
fn main() {
    let mut operationNumber = 1;

    // Operation 1: Declare/create an empty list
    println!("\nOperation {}", operationNumber);
    let mut languages = vec![];
    printResultingList(&languages, operationNumber);
    operationNumber = operationNumber + 1;

    // Operation 2: Initialize a list with some values
    println!("\nOperation {}", operationNumber);
    languages = vec![
        "Dart".to_string(),
```

```
        "Go".to_string(),
        "Javascript".to_string(),
        "Lua".to_string(),
        "Python".to_string(),
        "Ruby".to_string(),
        "Rust".to_string(),
    ];
    printResultingList(&languages, operationNumber);
    operationNumber = operationNumber + 1;

    // Operation 3: Check if the list is empty or not
    println!("\nOperation {}", operationNumber);
    isListEmpty(&languages);
    printResultingList(&languages, operationNumber);
    operationNumber = operationNumber + 1;

    // Operation 4: Add a new element to a list
    println!("\nOperation {}", operationNumber);
    let newElement = String::from("SQL");
    languages.push(newElement.clone());
    printResultingList(&languages, operationNumber);
    operationNumber = operationNumber + 1;

    // Operation 5: Check if a particular element exists in the list
    println!("\nOperation {}", operationNumber);
    doesContain(&languages, &newElement);
    printResultingList(&languages, operationNumber);
    operationNumber = operationNumber + 1;

    // Operation 6: Remove a particular element from the list
    println!("\nOperation {}", operationNumber);
    languages.retain(|item| *item != newElement);
    printResultingList(&languages, operationNumber);
    operationNumber = operationNumber + 1;

    // Operation 7: Get the head and the tail of a list
    println!("\nOperation {}", operationNumber);
    println!("Head of the list is: {}", languages[0]);
    println!("Tail of the list is: {}", &languages[1..].join(" "));
    printResultingList(&languages, operationNumber);
    operationNumber = operationNumber + 1;
```

```
    // Operation 8: Print all of the elements in the list
    println!("\nOperation {}", operationNumber);
    printResultingList(&languages, operationNumber);
    operationNumber = operationNumber + 1;
}

fn isListEmpty(list: &[String]) {
    if list.is_empty() {
        println!("List is empty");
    } else {
        println!("List is not empty");
    }
}

fn doesContain(list: &[String], object: &String) {
    if list.contains(object) {
        println!("List contains {}", object);
    } else {
        println!("List does not contain {}", object);
    }
}

fn printResultingList(list: &[String], operationNumber: i32) {
    println!(
        "List after Operation {}: {}",
        operationNumber,
        list.join(" ")
    );
}
```

1.7.2. Execution Result

```
Operation 1
List after Operation 1:

Operation 2
List after Operation 2: Dart Go Javascript Lua Python Ruby Rust

Operation 3
List is not empty
```

```
List after Operation 3: Dart Go Javascript Lua Python Ruby Rust

Operation 4
List after Operation 4: Dart Go Javascript Lua Python Ruby Rust SQL

Operation 5
List contains SQL
List after Operation 5: Dart Go Javascript Lua Python Ruby Rust SQL

Operation 6
List after Operation 6: Dart Go Javascript Lua Python Ruby Rust

Operation 7
Head of the list is: Dart
Tail of the list is: Go Javascript Lua Python Ruby Rust
List after Operation 7: Dart Go Javascript Lua Python Ruby Rust

Operation 8
List after Operation 8: Dart Go Javascript Lua Python Ruby Rust
```

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

An empty vector is initialized with an empty vector literal where vec! is a macro for Vec::new() function and mut is for mutable.

1.7.3.2. Operation 2

A vector is initialized with a vector literal where vec! is a macro for Vec::new() function call and they are semantically the same. Moreover string literals with &str type are converted to String objects with *to_string* function of str.

1.7.3.3. Operation 3

The vector object is passed by reference with *&* operator into the *isListEmpty* function which accepts an vector variable. *isListEmpty* function that I defined uses the built-in *is_empty* function of vectors returning the bool value corresponding to emptiness status of the vector. Then the corresponding result is printed within an if-else block.

1.7.3.4. Operation 4

A new string object which is created by *from* function of String to convert str object to String object, is added to the vector with the built-in *push* function (*clone* function of String is utilized since objects are moved (borrowed) into calling functions and with clone any subsequent borrows won't interrupt the program as cloned data is borrowed priorly) belonging to vectors, which accepts an object as parameter.

1.7.3.5. Operation 5

The vector and a string object which holds a value identical to that of an object in the vector is passed by reference (With & operator) into the *doesContain* function which accepts an vector and an object variable. *doesContain* function that I defined uses the built-in *contains* function which accepts an object formal parameter and returns a bool value after checking if the vector includes the object or not. Then the corresponding result is printed within an if-else block and *println!* macro which prints the formatted string.

1.7.3.6. Operation 6

A string object which holds a value identical to that of an object in the vector is passed into the built-in *retain* function which accepts an vector object and an object to remove the elements with value identical to given object's value from the vector.

1.7.3.7. Operation 7

Head (first element) of the vector is accessed with the built-in *subscript operator* of the vector by accessing element at index 0. Tail (subvector excluding the first element) of the vector is accessed with the built-in *subscript slice operator* of the vectors which accepts two index values and returns the subvector between passed indexes. Tail is from the second element (index at 1) until the last element (Empty second index means until to end of the vector). Moreover the built-in *join* function of *strings* is used for converting the vector object to a string object with proper format (spaces between elements).

1.7.3.8. Operation 8

The vector object is passed into the *printResultingList* function which accepts an vector and an operationNumber variable. *printResultingList* function that I defined which does string concatenation operations by *,* operator. Moreover, the built-in *join* function of vector which is used for converting the vector object to a string object with proper format (spaces between

elements). Then the concatenated string is printed to show the vector's final state.

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 list operation functionality into three categories:

First Category (+1 Points): Low level functionality with mediocre readability and writability in terms of Simplicity, Orthogonality and Syntax Design.

Second Category (+2 Points): High level functionality with good readability and writability in terms of Simplicity, Orthogonality and Syntax Design.

Third Category (+3 Points): Unique high level functionality with good readability and writability in terms of Simplicity, Orthogonality and Syntax Design.

2.1. Dart (Used) List Operations

Dart has a built-in list type as a mutable sequence allowing literal initialization.

2.1.1. Built-in List Properties (3 Points)

- *isEmpty* → Emptiness Checking Property
- *first*→ First Element Property

2.1.2. Built-in List Functions (10 Points)

- *add()*→ List Element Adding Function
- *contains()*→ List Inclusion Checking Function
- *remove()*→ List Element Removing Function
- *sublist()*→ List Slicing Function
- join() → List Formatting to String Function

2.1.3. Built-in List Operators (1 Point)

- = → Assignment & Initialization Operator

2.2. Go List Operations

Go has a built-in slice type imitating a mutable sequence allowing literal initialization.

2.2.1. Built-in List Properties (0 Point)

2.2.2. Built-in List Functions (6 Points)

- *append()*→ List Element Adding Function
- Join() → List Formatting to String Function
- len() → List Size Function

2.2.3. Built-in List Operators (4 Points)

- *:=* → Assignment & Initialization Operator
- [index] → List Subscript Access Operator
- [startIndex: endIndex] → List Slicing Operator

2.3. JavaScript List Operations

Javascript has a built-in array type as a mutable sequence allowing literal initialization.

2.3.1. Built-in List Properties (1 Point)

- *length*→ List Size Property

2.3.2. Built-in List Functions (13 Points)

- *push()*→ List Element Adding Function
- *includes()*→ List Inclusion Checking Function
- *indexOf()*→ List Element Retrieving Function
- *splice()*→ List Element Removing Function
- *slice()*→ List Slicing Function
- join() → List Formatting to String Function

2.3.3. Built-in List Operators (2 Points)

- *:=* → Assignment & Initialization Operator
- [index] → List Subscript Access Operator

2.4. Lua List Operations

Lua has a built-in table type as a mutable sequence allowing literal initialization.

2.4.1. Built-in List Properties (0 Point)

2.4.2. Built-in List Functions (10 Points)

- *insert()*→ List Element Adding Function
- *next()*→ List Iteration & Emptiness Checking Function
- *remove()*→ List Element Removing Function
- unpack()→ List Slicing Function
- concat() → List Formatting to String Function

2.4.3. Built-in List Operators (2 Points)

- = → Assignment & Initialization Operator
- [index] → List Subscript Access Operator

2.5. Python List Operations

Python has a built-in list type as a mutable sequence allowing literal initialization.

2.5.1. Built-in List Properties (1 Point)

- *listName* → Emptiness Checking Property (If used directly in a conditional expression)

2.5.2. Built-in List Functions (6 Points)

- *append()*→ List Element Adding Function
- *remove()*→ List Element Removing Function
- join() → List Formatting to String Function

2.5.3. Built-in List Operators (6 Points)

- = → Assignment & Initialization Operator
- *in* → List Inclusion Checking Operator
- [index] → List Subscript Access Operator
- [startIndex: endIndex] → List Slicing Operator

2.6. Ruby List Operations

Ruby has a built-in array type as a mutable sequence allowing literal initialization.

2.6.1. Built-in List Properties (3 Points)

- *empty?* → Emptiness Checking Property
- *first*→ First Element Property

2.6.2. Built-in List Functions (10 Points)

- *push()*→ List Element Adding Function
- *include?()*→ List Inclusion Checking Function
- *delete()*→ List Element Removing Function
- *slice()*→ List Slicing Function
- join() → List Formatting to String Function

2.6.3. Built-in List Operators (1 Point)

- = → Assignment & Initialization Operator

2.7. Rust List Operations

Dart has a built-in list type as a mutable sequence allowing literal initialization.

2.7.1. Built-in List Properties (1 Point)

- *first*→ First Element Property

2.7.2. Built-in List Functions (11 Points)

- *push()*→ List Element Adding Function
- *is_empty()*→ Emptiness Checking Function
- *contains()*→ List Inclusion Checking Function
- *retain()*→ List Element Removing Function
- join() → List Formatting to String Function

2.7.3. Built-in List Operators (4 Points)

- = → Assignment & Initialization Operator
- [index] → List Subscript Access Operator
- [startIndex.. endIndex] → List Slicing Operator

2.8. Language Evaluation Conclusion and Table

Simplicity Criteria / Languages	Dart	Go	JS	Lua	Python	Ruby	Rust
Built-in List Properties	3	0	1	0	1	3	1
Built-in List Functions	10	6	13	10	6	10	11
Built-in List Operators	1	4	2	2	6	1	4
Total Points	**14**	**10**	**16**	**12**	**13**	**14**	**16**

According to the evaluation process two languages reside at the top namely **Javascript** and **Rust** which both share the same final total point. Since the languages' readability and writability are evaluated in terms of their list manipulation operations and the operations are limited with the usage on the given tasks, these points may not reflect the readability and writability comparison of all coverance of the languages. However the whole code segments can be used as a tiebreaker between Javascript and Rust besides the evaluation of list operators. Hence when the overall readability and writability of Javascript is significantly greater compared to Rust since Rust contains many macros for simple operations, multiple String types which need to be converted to each other frequently, unconventional initialization operations, and unconventional borrowing mechanisms which forces programmer to use many pass by reference and clone functions. In conclusion none of these problems occur in Javascript while list manipulation operations are very strong. Therefore Javascript is the best programming language when it comes to list operations among all of the 7 languages used in the report

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.