Programming Paradigms – Loops and Iteration

The task at hand is to analyse how loops and iteration can be used to solve programming tasks in a programming language that is new to me and compare it to languages that have been taught through the course of the module. The programming language that was chosen for analysis in this assignment is "Go" and it will be compared against Java and Ruby. Whilst comparing programming languages it is important to cover their differences in three categories: syntax, semantics and pragmatics; these are defined below.

Syntax

"*The structure of statements or elements in a computer language."* (Dictionary.cambridge.org, 2019)

It is essentially how a programming language requires the code to be structured so that the compiler understands the instructions and can therefore execute them. One can think of syntax in two different ways. There's concrete syntax and abstract syntax. Concrete syntax considers the rules that apply when structuring the text typed into the editor, this differs from language to language. Abstract syntax is the rules that structure the conceptual idea behind a language feature, in this case loops and iteration. Abstract syntax (in general) doesn't differ between languages, however, some languages may support the use of a particular conceptual idea whereas others won't. For example, the "do…until" concept is allowed in Ruby but not Java (although Java does allow "do…while" which can be used to solve similar types of problems but "while" and "until" are opposites, so the concepts differ).

Semantics

"*Semantics is how we assign meaning to the well-formed phrases of a language. In human languages meaning is often deliberately vague, for example, how would you define the meaning of "game"? In computer languages, meaning must be very precise*" (Lane, 2017).

My understanding of semantics is that it is a description of how the computer takes a particular language feature or keyword that is used by the programmer to specify how they want the computer to behave and uses (usually) the compiler to translate that feature or keyword into binary values that can be understood by the machine and executed. Two popular ways of describing semantics are:

1. Denotational Semantics – each piece of syntax is assigned a well-defined mathematical form, which therefore gives it "meaning" that can be understood by the machine.
2. Operational Semantics – the "meaning" of a program is described as a series of instructions/operations on an abstract machine. Done using two stacks: values and control stacks. The control stack is used to store the expressions that are to be evaluated by the compiler. The values stack stores values which have been obtained by evaluating such expressions. The computer is programmed through determining what to do based on what is on top of both stacks.

Pragmatics

When used in a computer science context pragmatics is concerned assessing the skills used to solve the task at hand (like use of design principles), efficiency and robustness of the code. This is largely dependent on the programmer rather than the way that a programming language works. Different programmer equals different pragmatics.

Tasks completed for comparison

1. Add up all the integers from a given start value, less than a given end value, incrementing by a given step value, storing the total in a variable and displaying its value. For example, with 10 as a start value, 20 an end value, and 2 as the step, the total would be 10+12+14+16+18=70.
2. Store a series of numbers in a data structure, such as an array or list or equivalent in your language. Show how to compute the total value of all the numbers in the data structure.
3. Repeat the above exercise, but this time compute the total of the even numbers only.

Loops and iteration – what are they?

Definition: "*Iteration, in the context of computer programming, is a process wherein a set of instructions or structures are repeated in a sequence a specified number of times or until a condition is met. When the first set of instructions is executed again, it is called an iteration. When a sequence of instructions is executed in a repeated manner, it is called a loop*." (Techopedia.com, 2019)

I understand it as, repeating set of operations that are automated through an algorithm to avoid code duplication and improve efficiency. In Go the only loop construct available to the programmer is "for", although it can be used in various ways.

The syntax:

```
for [condition |  ( init; condition; increment ) | Range] {
   statement(s);
}
```

Source: (www.tutorialspoint.com, 2019)

You can use it in three ways. With a condition which is a Boolean expression, this executes while the condition is true. With three parameters controlling the loop, "init" being the definition of the loop control variable and its start value, "condition" being a Boolean condition which when evaluated to true takes the loop to the "increment" statement, this allows you to change the control variable on every iteration. The "init" and "increment" parameters can be left blank (as long as semicolons are there). The third way of using the "for" loop is the "range", this executes the loop for each item in the range, used for arrays.

Task 1 Go

```
 1 package main
 2
 3 import "fmt"
 4
 5
 6 func main() {
 7     var startVal int
 8     var endVal int
 9     var stepVal int
10     var total int
11     fmt.Println("Enter Start Value: ")
12     fmt.Scanf("%d", &startVal)
13     fmt.Println("Enter End Value: ")
14     fmt.Scanf("%d", &endVal)
15     fmt.Println("Enter Step Value: ")
16     fmt.Scanf("%d", &stepVal)
17     if (startVal < endVal){
18
```

```
19          for i := startVal; i < endVal; i = i + stepVal {
20              total = total + i
21              fmt.Print(i)
22              if (i + stepVal < endVal){fmt.Print(" + ")}
23          }
24      } else{
25        fmt.Println("Start value is larger than end value!")
26      }
27      fmt.Println(" = " , total)
28
29  }
```

Analysis

Lines 7-10 is a set of variable declarations. The “var” statement tells the compiler that this is a variable, then followed by the identifier for the variable, and data type which in this case is “int” (integer). The data type is optional, since Go allows the programmer to dynamically declare variables, which makes the compiler infer the data type of the variable based on evaluating the value passed to it.

Lines 11-16 use functions from the “fmt” package imported at line 3. The “Println()” function takes a string as a parameter and outputs that string to the console. The “Scanf()” is used for user input, the string parameter “%d” indicates that the input is to be an integer, the comma separates the two parameters and the “&startVal” tells the function where to store the user input.

Line 17 is a branching construct “if” (two-way). After the “if” key word is a Boolean expression enclosed by brackets, followed by a block of code enclosed by “{}”. The code enclosed by “{}” is executed only if the Boolean expression is true. If statements can be used for iteration as well, this will be illustrated in an example further on.

Line 19-22 illustrates the second use of a for loop described above, with these as parameters: `init; condition; increment`. The “init” is “i” which gets assigned “startVal” to it on the first run of the loop. The loop repeats while the “startVal < endVal” condition is true and it increments by “stepVal” on every repetition. The code within the block adds “i” to total, prints “i” to the console and adds a “+” sign (if “i + stepVal < endVal” is true) to illustrate what is happening to the user.

Line 24 – 26 is a part of the if statement (started at line 17). The “else” keyword indicates what happens if the “startVal < endVal” condition is evaluated to false, the block of code that is executed when that happens is again enclosed by “{}”. In this case it again uses “Println() function from the “fmt” package to output a string (an error message) to the console.

Line 27 again uses “Println()” which prints a string concatenated with the value of “total” (a comma separating the two) to the console.

In summary, the above example takes 3 values from the user: startVal the value at which the program starts counting, endVal the value at which the program stops counting and stepVal which tells the program the amount to add to increment every loop. The if statement checks whether the “startVal” is less than “endVal” for error prevention. If the condition is met the program uses a for loop which runs the block of code from the start value to the end value incrementing by the step value each time. Every increment the “i” is being added to the “total” variable and printed on the same line, a “+” sign is printed on that same line as long as “i + stepVal < endVal” is true. When the

condition in the for loop turns to false the program prints out the integer value stored in “total” to the console.

Compared to Ruby

Task 1 Ruby

```ruby
puts "Enter Start Value: "
startVal = gets.to_i
puts "Enter End Value: "
endVal = gets.to_i
puts "Enter Step Value: "
stepVal = gets.to_i
total = 0
if startVal < endVal
  for i in (startVal..endVal).step(stepVal)
    if i < endVal
      print "#{i}"
      total = total + i
      if (i + stepVal) < endVal
        print " + "
      end

    end
  end
else
  puts "its higher"
end
puts " = #{total}"
```

In terms of abstract syntax, the two examples are almost identical. The difference is in concrete syntax. In Ruby “puts” is used as the key word for a string to be output to the screen.

In Go the “Scanf(“%d”, &startVal)” function is passed two parameters: a string which indicates a datatype that the user input is to be cast to and a variable in which to store the user input. There is no need to assign the result of the function to a variable as the function does it for us, hence why it asks for the second parameter. In ruby the result of the function “gets.to_i” (right side of equals sign) is assigned to startVal (left side of the equals sign). This is because the user input is the result value of “gets.to_i” function and this result needs to be stored somewhere in memory, the function doesn’t do that automatically for us like in Go. The “to_i” part is necessary to make sure that the input is an integer, notice how this is a method in ruby whereas in Go the type is indicated by the first parameter passed to “Scanf()”, “to_i” is not necessary but helps the compiler avoid making mistakes when trying to infer the data type.

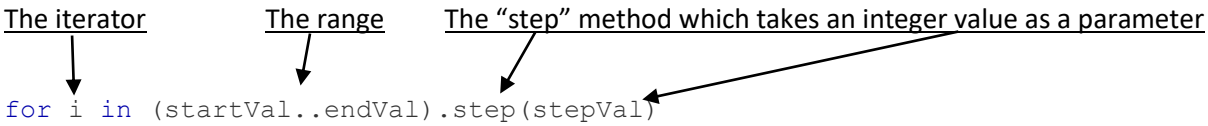
The syntax for the if statement is also different as the Boolean expression is not enclosed by brackets like in Go, also the block of instructions that are to be executed if the condition is true starts on the line after the if statement and the end of the block is indicated by the “end” key word instead of “{}” like in Go. This is the same for any loop construct available in Ruby.

The “for” loop syntax Ruby:

```
for variable [, variable ...] in expression [do]
   code
end
```

Source: (www.tutorialspoint.com, 2019)

How it's used in this example:




```
for i in (startVal..endVal).step(stepVal)
    if i < endVal
      print "#{i}"
      total = total + i
      if (i + stepVal) < endVal
        print " + "
      end

    end
```

The outcome of this loop is the same as the Go example however it is achieved slightly differently because of the syntax differences between the two languages. After the "for" key word the identifier for the variable that will be the iterator is required. Then the "in" key word indicates to the compiler that the content of the brackets is the range in which to operate. If left like this the iterator would increment one by one, however the task was to implement a step which gets added to the iterator every time the loop runs. There is a method which achieves that for the programmer called "step()" which takes an integer value as a parameter in this case "stepVal". Since Ruby uses a range for the loop (which can be done in Go as well however for this example it was more efficient to use the `init; condition; increment` version, this one does not exist in Ruby) there needed to be an if statement making sure that the value of "i" only gets added to the total when "i < endVal" is true. Otherwise, the value of "i" when "i = endVal" is true would be added to the total as well, this would produce a wrong result.

Compared to Java

Task 1 Java

```
1 public static void main(String[] args) {
2
3         Scanner userInput = new Scanner(System.in);
4         System.out.println("Enter Start Value: ");
5         int startVal = userInput.nextInt();
6         System.out.println("Enter End Value: ");
7         int endVal = userInput.nextInt();
8         System.out.println("Enter Step Value: ");
9         int stepVal = userInput.nextInt();
10        int total = 0;
11        if (startVal < endVal){
12            for(int i = startVal; i < endVal; i = i + stepVal){
13
14                total = total + i;
15                System.out.print(i);
16                if(i + stepVal < endVal){System.out.print(" + ");}
17
18
19            }
20        }else{
21          System.out.println("Start value is larger than end value!");
22        }
```

```
23          System.out.println(" = " + total);
24      }
```

In Java like in Go, before one can start thinking about getting the input from the user they must import a package, in Java it's done like this: `import java.util.*;` . Here you can also see that any statement in Java must be ended with a semi-colon, in Go that only occurs in the for loop to separate the parameters, the Go compiler puts a semi-colon at the end of each statement during runtime.

"Scanner" is a class from the "java.util" package. The `Scanner userInput = new Scanner(System.in);` statement is creating a new instance of that class which has functions that can be used to retrieve user input and return them as a value when called (`int startVal = userInput.nextInt();`). The difference between the two languages is that Go doesn't require the programmer to create an instance of the class, it retrieves the user input by just calling the function and passing the right parameters and the rest is done for you like so: `fmt.Scanf("%d", &startVal).` In Java the result of the function must be assigned to a variable of the same type similar to Ruby's example. Also, notice that every variable in java when defined must be given a type in this case "int" (Integer), the compiler in Java cannot infer the data type from the value assigned to it like in Go or Ruby.

`System.out.println("Enter Start Value: ");` is used as the statement that outputs a string to the user on a new line. Although the key word "System.out." is different to the Go key word and the semi colon in Go is not necessary, the structure of the statement is somewhat similar as both languages require brackets and speech marks to surround the string which will be output to the user.

The concrete syntax for the conditional construct "if" is identical in both languages. The "for" loop concrete syntax is similar but differs slightly. The only difference is that in Java's case brackets need to surround the control values, where Go doesn't require that. Also, when the increment value is initialised Java requires a type whereas, Go's compiler is forced to infer it from the value being assigned to it using the ":=" operator instead of "=". Other than that, they're identical.

The last slight difference between the two is that when a string is to be concatenated with a variable a comma is used to separate the two in Go, in Java it's the plus operator.

Task 2 Go

```
 1 package main
 2 import "fmt"
 3 func main() {
 4   var numbers []int
 5   var userInput int
 6   var arraySize int
 7   var index int = 0
 8   var total int = 0
 9   fmt.Println("How many numbers do you want to store in the array ?")
10   fmt.Scanf("%d", &arraySize)
11   for index < arraySize {
12       fmt.Scanf("%d", &userInput)
13       numbers = append(numbers, userInput)
14       index++
15   }
16   for _, number:= range numbers{
```

```
17      fmt.Println(number)
18      total = total + number
19    }
20    fmt.Println(total)
21 }
```

Analysis

Note that all concepts described previously that appear in this example have been omitted.

Line 4 – Due to the fact that in Go arrays cannot have a size which is a variable (only constants) I had to find an alternative. This alternative is what you call a "slice" in Go terms which in this case is called "numbers". A slice is essentially a dynamic array which gets automatically resized when an item is appended to it

Lines 11-15 is a for loop using the following syntax:

```
for [condition] {
   statement(s);
}
```

The for loop repeats while the "index < arraySize" expression evaluates to true. Within the block defined by "{}" the user input is retrieved and assigned to the "userInput" variable. That user input is then added to the slice described above using the: `numbers = append(numbers, userInput)` statement. Append is an in-built function which takes a slice (in this case "numbers") and element (in this case "userInput") as parameters. The function then returns a new, dynamically resized slice as its result, therefore the result of the "append" function needs to be assigned to "numbers" on the right side of the "=" operator. The final statement in the block is a simple increment of the index variable.

Lines 16-19 is a for loop using the following syntax:

```
for [index, element:= range slice[]] {
   statement(s);
}
```

In the example the range is anonymous indicated to the compiler by the use of "_", if an identifier was used for the index value, that value would have to be used somewhere within the scope of the loop (inside the "{}" enclosed block), otherwise it wouldn't compile throwing an error which indicates that the value was declared but not used, making it anonymous gets rid of that issue. The element parameter in this example is "number" followed by the ":=" operator. This statement tells the compiler that every increment the current slice element is to be assigned to "number", the ":=" operator forces the compiler to infer the data type of the element, as it is not defined in code. The "numbers" slice indicates to the compiler that the for loop is to loop through each element stored in the slice. Within the block enclosed by "{}" there's two statements, the first outputs the current element stored in "number" to the console, the second adds that element to "total" which (once the loop is completed) holds the sum of all elements within the slice.

Line 20 – just outputs "total" to the console.

In summary, the code initialises an empty slice which is typed as an integer. Then, the code defines "userInput" and " arraySize" variables as integers. The "arraySize" variable is for holding the size of the slice which is input by the user. The "userInput" variable is used to hold the element that the user wants to add to the slice. The "index" and "total" variables are to hold the increment variable, and the sum of all slice elements respectively. The user is asked how many elements they want the slice to contain, the value that the user types is assigned to "arraySize". The "for" loop then runs

while the “index < arraySize” expression is true. Every time the code in the block defined by “{}” runs the user is prompted for the element to be added to the slice, this element is assigned to the “userInput” variable. The next line uses the in-built “append” function to add the element stored within “userInput”. The “index” is incremented by one. When the condition “index < arraySize” turns to false, the compiler breaks out of the loop and enters another “for” loop, this time it’s a “for” loop which iterates over every element in “numbers[]”. Every iteration it and stores the current element in “number”. The body of the loop instructs the compiler to output “number” to the console, then add it to “total”. When the loop is finished the total gets output to the console which is equivalent to all the slice’s elements added together.

Compared to Ruby
Task 2 Ruby

```
 1 puts "How many numbers do you want to store in the array?"
 2 arraySize = gets.to_i
 3 numbers = Array.new(arraySize)
 4 total = 0
 5 index = 0
 6 while index < numbers.size do
 7   numbers[index] = gets.to_i
 8   index +=1
 9 end
10
11 numbers.each do |number|
12   puts number
13   total = total + number
14 end
15 puts total
```

Note that all comparisons described previously that appear in this example have been omitted.

Ruby allows variables to be used as the size of an array, although it cannot be resized after it’s declared. Therefore, in this example an array is used as the data structure, it is declared at line 3. To declare an array in Ruby, the programmer must assign the result (an empty array) of the “Array.new(<sizeOfArray>)” function to a variable(in this case “numbers”). Since the result of the just mentioned function is an array type the, Ruby compiler will infer the type from this result, therefore, turning “numbers” into an array.

Lines 6-9 are a while loop construct that is available in Ruby. The concrete syntax of this loop differs from that of Go. Instead of “for” Ruby uses “while” as the key word to start this loop, it is followed by a Boolean expression (same as Go). Ruby then starts the block which contains the statements that are to be executed whilst the condition is true with “do” and closes it with “end”, in Go the statements are enclosed in “{}”. However, the abstract syntax is the same. The loop checks the Boolean expression, while the result is true it executes the block of statements present in the body of the loop, it ends when the expression evaluates to false.

Since in Ruby I used an array to implement the solution the result of the user input can just be assigned directly to the array element “numbers[index]”, instead of using “append” and assigning the new resized array as the new value of “numbers”.

Lines 11-14 are a demonstration of the in-built “each” function that can be called on any array in Ruby. To use it the `numbers.each do |number|` statement is used, “numbers.each” indicates to the compiler that “each” should be used on the previously declared “numbers” array. The “do |number|” part tells the compiler that each element that is retrieved every time the construct loops

should be assigned to a new variable called “number”. The closure of the block of statements is again indicated by the “end” key word.

Compared to Java
Task 2 Java

```
 1  public static void main(String[] args) {
 2         System.out.println("How many numbers do you want to store in
 3 the array ?");
 4         Scanner userInput = new Scanner(System.in);
 5         int arraySize = userInput.nextInt();
 6         int numbers[] = new int[arraySize];
 7         int total = 0;
 8         int index = 0;
 9         while (index < numbers.length){
10             numbers[index] = userInput.nextInt();
11             index++;
12         }
13         for (int number: numbers){
14             System.out.println(number);
15             total = total + number;
16         }
17         System.out.println(total);
       }
```

Note that all comparisons described previously that appear in this example have been omitted.

The `int numbers[] = new int[arraySize];` (line 5) statement is an instantiation of the array class which is of the integer type. This is not a function rather a creation of the Array object. The type “int” is present on both sides of the “=” operator as the container for the array (left side) and the “new” instance of the array (right side) must be the same. The “[]” on the left side are there to indicate to the compiler that the container is to be of the array type, the size of the array “arraySize” is in “[]” on the right side. This means that the new instance of the Array class has a size which is the value stored in the “array size” variable. When the compiler evaluates this statement, a new empty array called “numbers” will be created with space for the number of elements specified by the value of “arraySize”.

Once again, the “while” loop construct (lines 9-12) differs in concrete syntax to the one used in the Go example. This time the difference is the “while” key word instead of “for”, the semicolons at the end of statements within the block and the Boolean expression which is to be evaluated before each iteration is enclosed by brackets in Java. Abstract syntax for this construct is identical to the one in Go.

The “for” loop (lines 13-16) in this again uses the “for” keyword to describe the type of loop to the compiler. Within the brackets is a declaration of a new variable “number” of type integer, this variable is to store the content of the current array element. The semi-colon is used to indicate to the compiler that the identifier that follows is the array that needs to be iterated through.

Task 3 Go

```
1  package main
2 import "fmt"
3 func main() {
```

```
 4   var userInput int
 5   var arraySize int
 6   var numbers []int
 7   var index int = 0
 8   var total int = 0
 9   fmt.Println("How many numbers do you want to store in the array ?")
10   fmt.Scanf("%d", &arraySize)
11   for index < arraySize {
12       fmt.Scanf("%d", &userInput)
13       numbers = append(numbers, userInput)
14       index++
15   }
16   for _, number:= range numbers{
17     if (number%2 == 0){
18       fmt.Println(number)
19       total = total + number
20     }else{
21       fmt.Println(number, " is not even")
22     }
23   }
24   fmt.Println(total)
25 }
```

Analysis

As you can see, the only difference in this example is the content of the second “for” loop which sums up all the elements in the data structure, this time there is an if statement which executes the “true” block only when the “number%2==0” expression evaluates to true. The “%” operator is used get the remainder of “number/2”. If the remainder is 0 that means that the current element stored in “number” is an even number and should be added to “total”, otherwise a message informing the user that the current element is odd, and it is not added to “total” is output to the console. The last thing to note is that Go provides the programmer with a “break;” statement which when read by the compiler indicates to it that the loop is finished even though the requirements to finish the loop might not have been met. This statement is also present in Ruby and Java. It helps the programmer control the flow of the program.

Comparison to Ruby & Java

The differences between “if” statements have already been covered. Also, in both Ruby and Java the “%” operator has the exact same effect as it does in Go. Since nothing else has been changed the following examples are not the full program but just what changed between example 2 and 3 within the “for” loop responsible for summing up the integers.

Ruby:

```
numbers.each do |number|

  if number % 2 == 0
    puts number
    total = total + number
  else
    puts "#{number} is not even"

  end
end
```

Java:

```
for (int number: numbers){
            if (number%2 == 0){
                System.out.println(number);
                total = total + number;
            }else{
                System.out.println(number + " is not even");
            }
}
```

One might wonder what if we wanted to code something like Java's "do…while" loop construct in Go. There is no such thing in Go, however, the effect can be imitated through the combination of the "if" branching construct and the "goto" statement which points to code specified by a label, to have a similar effect the label should be placed above the if statement so that the condition is tested once the statements enclosed have been executed one time already.

Summary and conclusion

In my opinion syntactically Go is more like Java than it is Ruby. Both use "{}" to enclose blocks of code, some similarities in the "for" loop syntax, similar method of declaring variables.

However, semantically Go's compiler reminds me more of Ruby's than Java's. This is because the variable types can be dynamically inferred by the compiler, through evaluating the value assigned to the variable, whereas, in Java the type of any variable must be specified before compilation (more static than dynamic).

Pragmatically, out of the three languages I feel like Java offers the most flexibility in style of programming and the most control over efficiency of the code and its robustness.

References

www.tutorialspoint.com. (2019). *Go for Loop*. [online] Available at: https://www.tutorialspoint.com/go/go_for_loop.htm [Accessed 23 Jun. 2019].

Lane, P. (2017). *Programming Paradigms Module Notes for 6COM1045*. p.5.

www.tutorialspoint.com. (2019). *Ruby Loops*. [online] Available at: https://www.tutorialspoint.com/ruby/ruby_loops.htm [Accessed 23 Jun. 2019].

Dictionary.cambridge.org. (2019). *SYNTAX | meaning in the Cambridge English Dictionary*. [online] Available at: https://dictionary.cambridge.org/dictionary/english/syntax [Accessed 22 Jun. 2019].

Techopedia.com. (2019). *What is Iteration? - Definition from Techopedia*. [online] Available at: https://www.techopedia.com/definition/3821/iteration [Accessed 22 Jun. 2019].