

An Evaluation Paper on Go using
Philippine Tax Calculation as the Application Domain

Submitted
in partial fulfillment of the requirements
for the course CSADPRG

Dichoso, Aaron Gabrielle C.
Tordillo, Christian Dave
Natividad, Josh Austin
Razon, Luis Miguel

Mr. Edward Tighe
12/04/2023

1. Introduction

1.1. Background of the Study

The features and attributes of a programming language must be taken into consideration whenever developing an application. This is because some of these features may actually hinder or benefit the development of the application, in terms of the readability and writeability of the code, the speed of the calculations, and the sustainability and accessibility of the application. For our purposes of the creation of a tax calculator, 4 programming languages were first considered. These programming languages are Ruby, R, Kotlin, and Go.

1.2. The Ruby Programming Language

Ruby was created in Japan by Yukihiro Matsumoto, whose vision was to create a language that was more powerful than Perl and more object-oriented than Python (*About Ruby*, n.d). By combining parts of Matsumoto's favorite programming languages, like Perl, Smalltalk, Eiffel, Ada, and List, he was able to develop Ruby in 1995.

Ruby is an interpreted language, which means that programs do not need to be compiled to be run and that scripts are able to execute code line by line. It is a purely object-oriented language with concepts derived from functional and imperative programming paradigms. Everything is an object in Ruby, wherein everything can be given its own properties and actions. For example, variables are objects of a class of their respective data types and operators are method calls on objects. This grants Ruby great flexibility because it allows even essential parts to be altered where even its operators could be removed or redefined (*About Ruby*, n.d.).

As Ruby evolved, many new features were added, such as (Castello, n.d):

1. The introduction of UTF-8 as its default encoding standard in Ruby 2.0
2. The introduction of Just in Time (JIT) Compiling as an experimental feature in Ruby 2.6, allowing Ruby applications to be compiled and still able to be run line by line; and
3. The introduction of Ractors, an alternative to threading in Ruby for concurrency and Garbage Collector improvements in Ruby 3.0

Currently, there is Ruby 3.2, which was released on December 25, 2022, and introduced the *Data* class, which serves to contain value objects intended to be immutable. Ruby has been said to have a strong and supportive community that is friendly towards people of all skill levels. It is currently being used in simulations, 3D modeling, business, robotics, telephony, system administration, web applications, and security testing (*Ruby Community*, n.d). Its wide array of tools and features allows it to be a potential candidate to be used for the creation of the tax calculator.

1.3. The R Programming Language

R was established in 1995 by Robert Gentleman and Ross Ihaka, both faculty members at the University of Auckland, as an open-source project based on the S programming language. R is designed with statistical computing in mind, allowing those who do not have access to the S programming language to still be able to perform statistical analyses since it is open-source. It supports functional, object-oriented, and procedural programming paradigms, with a heavy emphasis on functional programming. (*What is R*, n.d; *What is R? MRAN*, n.d).

Over the years, R has gained popularity amongst data scientists, researchers, and statisticians, and became a widely used programming language for statistical computing or data analysis (*What is R*, n.d).

R supports multiple programming paradigms, however, it primarily supports functional programming. Functional programming emphasizes the use of immutable data structures and functions to perform tasks. With that in mind, R also supports object-oriented programming, and procedural programming (*What is R*, n.d).

R uses higher-order functions, closures, and anonymous functions to perform calculations. The approach is well-suited for data analysis and statistical computing with a wide range of operations on huge data sets. And functions are first-class objects, which means that they can be assigned to variables, which allows for a lot of flexibility and can even allow for functions to be passed as variables (*What is R*, n.d).

R also supports OOP through the S3 and S4 object systems. S3 is a simple and flexible system for defining classes and methods in R while S4 is a more structured and formal system, both will allow you to define classes and create instances of those classes (*What is R*, n.d).

Throughout the years, R has had a total of 5 major releases, and below are its most notable new features.

1. Statistical Algorithms
2. A memory management system that improved the efficiency of large data sets.
3. Namespaces
4. Data Frames
5. Support for probabilistic programming
6. Updates to data handling and security

The latest major version release of R. Released in 2020, it too included improvements to performance and memory management, and support for new data types. The version included new features such as a new syntax for specifying character strings, enhanced support for data frames, and improvements to package development. For the future, the R Team plans to improve performance and scalability which includes improvements on the core engine, the optimizing packages, and developing possible new ways to approach parallelization, enhance data-handling capabilities, support for new data types like nested data frames and improvement on the handling

of large and complex data sets, expand visualization capabilities including new types of charts and visualizations at the same time upgrading already existing ones, increase support for Machine Learning and Artificial Intelligence which includes new algorithms and libraries and improving integration with other frameworks and tools, improve integration with cloud platforms (*What is R*, n.d).

1.4. The Kotlin Programming Language

Kotlin was developed by a company called JetBrains in 2010 and its first version was officially released in February of 2016. It has gone open-source early on within its development cycle and it was sponsored by Google, receiving first-class support, and announced as one of its official languages by 2017 (GeeksForGeeks, 2023; Lardinois, 2019), then becoming the most preferred language by Android developers since 2019 (Shafirov, 2019).

Kotlin is a statically typed programming language that combines all major programming paradigms. It allows the use of functional, imperative, object-oriented, and procedural programming concepts with others such as concepts of concurrency and parallelism (JetBrains, n.d.). This benefits Kotlin programmers because it allows them to develop applications with less complex code while being more expressive and inheriting the strengths of other well-known languages such as classes and object-oriented code that would be normally seen in Java, with less of its boilerplate code, and procedural code much like in JavaScript or C which allows for the execution of scripts without extra compilation but with added type safety features that comes with statically typed languages (Educative, n.d.).

Kotlin has greatly improved from its initial 1.0 release, some of these major improvements are (Belov, 2017; JetBrains, 2023):

1. Full Javascript support, meaning JS files can now be migrated to Kotlin applications while still allowing the use of modern JS frameworks such as React.
2. The introduction of coroutines which allows for more scalable application backends compared to threads.
3. Introduction of the Kotlin/JVM K2 compiler which brings performance improvements for JVM, JS, and Native platforms

The current release of Kotlin is the 1.8.0 release, specifically 1.8.20 which was released on April 3, 2023. Its biggest improvements was the introduction of the new Xdebug compiler option that would improve the debugging experience, improved Objective-C/Swift compatibility, and experimental Base4 encoding support in the standard library (JetBrains, 2023; JetBrains, 2023). In the future, Kotlin plans to rewrite the K2 compiler to be optimized for speed, parallelism, and unification and will also bring many new language features (JetBrains, 2023).

1.5. The Go Programming Language

Go was released by Google in late 2007 to address the problems of the company in its software infrastructure. Its development started on September 21, 2007, when three Google engineers, namely Robert Griesemer, Rob Pike, and Ken Thompson, wanted to develop a simple language with efficient compilation and execution, much like C++ but without its complexity. Go was designed and developed with productivity and robustness in mind with certain concepts such as native concurrency and garbage collection support as well as including rigorous dependency management, and adaptability to growing software architecture (Pike, n.d.). The syntax of Go is mostly inspired by the C family in order to focus on the simplicity of the language (Rybacka, 2021).

Go is a compiled and statically typed programming language. It mostly follows the imperative programming paradigm with some concepts being derived from the object-oriented paradigm. It contains loops, statements, and selections, and allows for types and methods in an object-oriented fashion. There is, however, no class hierarchy but there is support for interfaces. Methods are present and can be designed for all types and there is a lightweight version of objects in Go called structs that is visually similar to C rather than C++ and Java (Zhang, n.d.).

After the first stable release of Go back in 2012, there have been many improvements over the years, such as (*The Go Blog - The Go Programming Language*, n.d.):

1. Faster garbage collection and optimization for multi-core CPUs
2. Allowing the execution of goroutines parallel to each other.
3. Support for different OS such as Darwin and iOS 64-bit ARM architecture
4. Introduction of Fuzzing - a type of automated testing that would continuously manipulate inputs in order to find bugs (*Go Fuzzing - The Go Programming Language*, n.d.).

The current major release of Go, Go 1.19, has seen big improvements in the language with doc comments now supporting links, lists, and clearer heading syntax and a variety of performance and implementation improvements such as dynamic sizing of initial goroutine stacks among others (The Go Team, 2022). Currently, Go is being used in web services, utility applications for large data processing or one-off tasks, financing or banking, and cloud computing (Zharova, 2021).

1.6. Application Domain

After understanding the features and limitations of each programming language, it is then important to take note of the application domain. The tax calculator uses the monthly income or salary of a person as input. Afterward, it would compute for the needed monthly contributions for the Social Security System (SSS), Philippine Health Insurance (PhilHealth), and the Pag-IBIG Fund. These contributions are all funds that are non-taxable, entailing that they should not be included in the tax calculations. The taxable income should then be obtained, which is to be used to obtain the income tax, and the total deductions that a person would have to pay given their monthly income. The calculations for the SSS, PhilHealth, Pag-IBIG, and Income tax can

be obtained directly from government sources. The guideline for calculating these contributions relies on a table, entailing that our programming language should have control structures that would support the translation from the table to code. Additionally, these rules for calculations change every year, meaning that a programming language whose structure supports rapid development is highly favorable.

Taking the features of the discussed programming languages and the application domain into consideration, the group has created the tax calculator in Go, due to its simplicity to write code, its speed in compiling programs, its efficiency in resource allocation and management, and its rich native concurrency library.

2. Language Comparison

2.1. Programming Paradigm

Object-oriented paradigms allow for classes, objects, abstraction, polymorphism, inheritance, and encapsulation. These concepts help the paradigm by improving the writability and readability of code because it allows the user to reuse code through inheritance, grants the programmer flexibility through polymorphism ideas such as method overloading, and the use of classes and objects helps with efficiency as it helps in breaking down big problems into smaller subproblems (Kumari, 2023).

This can be clearly seen with Ruby, a purely object-oriented programming language. All of the object-oriented concepts are present in all aspects of Ruby. Classes and objects in Ruby are similar to Python where they are declared by a class name which can have instance variables that can be assigned through an *initialize* method. Instance variables can also be used in methods associated with the class. Inheritance is also present in Ruby where a class can inherit the methods and attributes of its parent class. Alongside inheritance, polymorphism can also be seen where the child class can alter the methods of its parent class to suit its own needs.

```

class Person
  def initialize(name)
    @name = name
  end

  def say
    puts("I am " + @name)
  end
end

class Child < Person
  def say
    puts ("I am a child and I am " + @name)
  end
end

p = Person.new("Rod")
c = Child.new("Henry")
p.say #prints "I am Rod"
c.say #prints "I am a child and I am Henry"

```

Figure 1.0 Ruby *Class declarations and Inheritance*

Other OO concepts such as abstraction and encapsulation are also present in Ruby. The use of access modifiers— such as private, protected, and public— are present in Ruby and it is how abstraction is achieved. Alongside access modifiers, classes also have getters and setters that control what the classes outputs and takes in.

These concepts are the backbone of Ruby. Everything in Ruby is an object of a class. All values are just objects of a class based on their own data type. Operators are just method calls from the object of the left operand and passing the right operand as its argument (Abhilash, 2023).

```

11.class.name #Integer
true.class.name #TrueClass
false.class.name #FalseClass
nil.class.name #NilClass

```

Figure 1.1 *Ruby Class Operation*

These features allow Ruby to have all the advantages of the object-oriented paradigm. Due to Ruby being a pure object-oriented language, it is highly flexible and expressive because with everything being an object, it is possible to modify how objects interact with each other by applying concepts of polymorphism and inheritance such as method or operator overloading. This can benefit readability and writability because the programmer can control the orthogonality of Ruby to suit the needs of their application.

However, because of its pure object-oriented paradigm, this can instead negatively affect readability and writability because it gives the programmer too much freedom, to the point where they can even remove and modify foundational operations, such as adding and subtracting integers. It can lead to too much orthogonality leading to unnecessary complexity which would make the program hard to understand since it would be hard to keep track of how different objects interact with each other.

Aside from Ruby, Kotlin also has support for object-oriented programming, as it is built to be able to interoperate with Java. In many ways, its object-oriented paradigm mirrors that of Java. Classes in Kotlin are declared with a class name and may contain a class header with possible type parameters and/or a primary constructor, a class body, functions, properties, and nested classes. (See Figure 2.0)

```
class Animal constructor(name : String, age : Int, type : String){ /*...*/ }
```

Figure 2.0 Kotlin Constructor

The primary constructor of a class being declared in the class header helps with readability as it allows for class constructors to be inline with the class name, simplifying the syntax. Primary constructors, however, do not contain any code. Instead, the initialization code is defined by an initialization block and the `init` keyword. A class can have multiple initialization blocks, which will be executed according to the order in which they appear in the class body. (See Figure 2.1)

```
class Animal constructor(name : String, age : Int, type : String)
{
    new *
    init {
        println("Hello! My name is: $name")
    }
}
```

Figure 2.1 Kotlin Class Body

Classes can also have secondary constructors, which can be added inside the class body. These constructors are called after initialization blocks and can be used to extend the functionality of a class. (See Figure 2.2)


```

class Animal constructor(name : String, age : Int, type : String)
{
    new *
    init {
        println("Hello! My name is: $name")
    }
    new *
    constructor(name : String, age : Int, type : String, numChildren : Int) : this(name, age, type)
    {
        println("I have $numChildren children!")
    }
}

```

Figure 2.2 *Kotlin Secondary Constructor*

All classes in Kotlin derive from a superclass, *Any*, which has 3 methods, *equals()*, *hashCode()*, and *toString()*, which are inherited and can be overridden by all other classes. By default, user-defined classes cannot be inherited from, and need to contain the *open* keyword for the class to be inheritable, as opposed to Ruby which does not have final classes as a standard implementation. This aids in the expressivity of the language, allowing classes to be manually specified by the user whether or not they are inheritable. Kotlin also expands upon the limitations of Java, allowing an object to inherit methods with the same name between a superclass and interface or other interfaces. (See Figure 2.3)

```

open class Mammal: Animal
{
    new *
    constructor(name : String, age : Int, type : String) : super(name, age, type)
    {
        sayType(type)
    }
    new *
    open fun sayType(type : String){}
}

new *
interface LaysEggs
{
    new *
    fun sayType(type : String){}
}

new *
class Platypus(name : String, age : Int, type : String) : Mammal(name, age, type), LaysEggs
{
    new *
    override fun sayType(type : String)
    {
        super<Mammal>.sayType(type)
        super<LaysEggs>.sayType(type)
    }
}

```

Figure 2.3 *Kotlin Class, Interface, and Inheritance declarations*

Aside from Kotlin's capabilities to support object-oriented programming, it also implements a functional paradigm into its design. Functional paradigms focus on the use of pure functions, which are functions that produce outputs based only on their inputs and cannot and do not modify any data outside of their scope. The approach places its emphasis on immutability, recursive functions, and higher-order functions which can then be used to create complex behavior from simple building blocks and it is particularly well-suited when it comes to tackling complex problems (GeeksForGeeks, 2023).

This type of programming is clear in R. However, R is not a pure functional programming language, and self-discipline is necessary to be able to apply the principles of functional programming (Rodrigues, 2023).

Higher order functions and immutability are important parts of functional programming. Higher-order functions are functions that can be used to create other functions, or rather, other generic and reusable code. Immutability refers to the concept of variables that, when assigned a value, cannot be changed, and that makes the code more predictable and easier to follow (FreeCodeCamp, 2023).

The R language supports higher-order functions by allowing for functions to be passed as arguments or be returned as values just like variables. Functions in R are considered as first-class objects, which means they are able to be assigned to variables, and that makes it possible to write higher-order functions. (Wickham, 2023).

The `lapply()` function in R is an example of a higher-order function. `lapply()` is a function that takes a function as its second argument and applies that function to each element of a list (the first argument). (See Figure 3.0)

```
1 my_list <- list(c(1, 2, 3, 4), c(5, 6, 7), c(7, 8, 9, 10))
2
3 result <- lapply(my_list, mean)
4
5 print(result)
```

Figure 3.0 R lapply() function as a higher-order function

Each item of the list has been turned into the value of the output of the `mean()` function given each item (See Figure 3.1).

```
[[1]]
[1] 2.5

[[2]]
[1] 6

[[3]]
[1] 8.5

[Execution complete with exit code 0]
```

Figure 3.1 List output of the R lapply() function

In terms of R's support for immutability, they do not have support for immutability. They do, however, have several packages available that provide the functionality. One of these packages is called the "immutable" package which makes way for immutable data structures for vectors and lists (R Project, 2023).

While Go does have the ability to simulate object-oriented programming, it can only do so to an extent. The language does have a method construct which is a function with a special receiver argument. This means that methods defined on specific types are only accepted if the corresponding type is passed to its special receiver argument (*A Tour of Go*, n.d.). The other principles of OOP, namely inheritance and polymorphism aren't supported, as the language does not have a class hierarchy (Makala, 2021).

A common ground among all the languages is that all four languages implement imperative and procedural programming paradigms. Imperative programming paradigm-based programs follow a clearly defined sequence of instructions to solve a problem. Based on this approach, these programs are coded as a series of statements that specifies what the computer has to do. They are controlled by using control structures such as loops or selection statements. This approach aids in writability and readability since a step-by-step sequence of instructions would be easy to write and understand but a clear disadvantage of the imperative paradigm is that it can take many lines of code in order to reach a goal (Ionos, 2020).

The procedural programming paradigm is a subset of the imperative paradigm which divides the sequence of instructions into named sets of instructions called procedures that can store local data and access or modify global data variables. This paradigm further aids the advantages of the imperative paradigm in readability and writability because it further supports code reusability which affects simplicity (Indeed Editorial Team, 2022).

While Go is a multi-paradigm language, it can be considered mainly a procedural language the way functions are created and used to perform procedures and steps executed by a computer to solve a specific problem (Paramitha, 2019). The language also provides the constructs imperative languages mostly use, such as loops, conditional statements, and selections, defined in its language specifications (Zhang, n.d.).

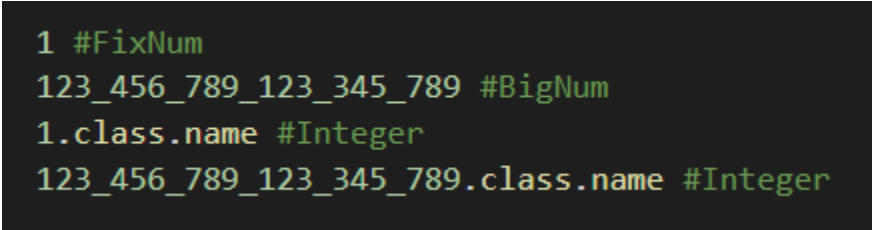
2.2. Data Types

For the purposes of the tax calculator application, the way in which data types are represented and handled by the programming language is a critical factor.

One of the key features of Kotlin is that everything is an object, which means that methods and properties of a variable can be used by the developer which in turn increases expressivity (JetBrains, 2023). In Kotlin, even primitive data types such as integers, strings, and boolean expressions are objects and can be treated as objects. They can be manipulated using the same syntax as objects. For real numbers, the language provides two floating-point types, Float and Double that uses the IEEE 754 standard for floating-point numbers and arithmetic (JetBrains, 2023). Unlike other languages, Kotlin does not allow implicit widening conversions for numbers. This means that a function that takes in a double type as a parameter will not accept types such as float and int. This constraint allows the language to prevent bugs regarding incorrect calculations and type mismatches (JetBrains, 2023).

In Ruby, all of its data types are classes. A value that is defined in a variable acts as a constructor where the variable automatically becomes an object of the class that corresponds to the type of the value. Basic types in Ruby are numbers, strings, ranges, arrays, hashes, symbols, and regular expressions (*Programming Ruby: The Pragmatic Programmer's Guide*, n.d.).

Numbers in Ruby are integers and floating-point numbers. Integers are objects of the FixNum class. If the integer is large enough, it automatically becomes an object of the BigNum class. FixNum and BigNum are however abstracted and instead displayed as objects of the Integer class. (See Figure 4.0)



```
1 #FixNum
123_456_789_123_345_789 #BigNum
1.class.name #Integer
123_456_789_123_345_789.class.name #Integer
```

Figure 4.0 Ruby Integer data types are objects

Strings in Ruby are objects of the String class. They are stored as sequences of 8-bit bytes. There are also string literals in Ruby such as *true*, *false*, and *nil*. They are objects of their own respective classes TrueClass, FalseClass, and NilClass respectively. This is exactly the reason why Ruby does not really have a “Boolean” data type even if a variable might contain true or false since the variable only becomes an object of the classes TrueClass or FalseClass which implies that it does not set a range of possible values unlike with the boolean data type whose range of values is true or false. (See Figure 4.1)

```
"Hello".class.name #String
true.class.name #TrueClass
false.class.name #FalseClass
nil.class.name #NilClass
```

Figure 4.1 Ruby Class Names

Ranges in Ruby are automatically created when an object uses the “..” or “...” method. It is a collection containing values in between the left and right values. (See Figure 4.2)

```
(1..4).each{|n| puts n} #1, 2, 3, 4
(1...4).each{|n| puts n} #1, 2, 3
(1...4).class.name #Range
('a'...'d').each{|n| puts n} #a, b, c
```

Figure 4.2 Ruby Range Modifiers

Arrays in Ruby are created by separating objects with a comma in between square brackets. Ruby uses heterogeneous arrays which means that its elements do not need to have the same type. It uses typeless references to access variables inside the array. (See Figure 4.3)

```
[1,2,3,4,5].class.name #Array
[1, 'b', 3, 1.4, 2].class.name #Array
```

Figure 4.3 Ruby array declarations

Hashes in Ruby are created by inserting key-value pairs in between braces, separated by a comma. The key and values can have different types but keys are most commonly strings. (See Figure 4.4)

```
{"John" => 12, "Paul" => 13}.class.name #Hash
```

Figure 4.4 Ruby hash declaration

Symbols in Ruby are considered as internal representations of a name. It is constructed by preceding the name with a semicolon. (See Figure 4.5)

```
:hello.class.name #Symbol
```

Figure 4.5 Ruby Symbols

Regular expressions in Ruby are objects of the Regexp class. They can only be created by calling Regexp.new or by using /pattern/ and %r{pattern}. (See Figure 4.6)

```
/pattern/.class.name #Regexp
```

Figure 4.6 Ruby Regular Expressions

Due to the nature of Ruby, all of the basic data types present are all objects of different classes. This is a strength of Ruby because it aids in writability through orthogonality. The different objects can interact with each other with less risk of errors because of built-in type checking and implicit conversion through coercion. However, the weakness of having all data types implemented as objects is that it negatively affects efficiency and storage since it would take more memory compared to primitive data types that are not object-based. For the tax calculator application, the strength of data types in Ruby would not contribute much because most, if not all, of the values that are being handled by the application are all decimals or floating point numbers so it does not need that much orthogonality. This would actually impact the application negatively because the strengths would not outweigh the weaknesses, making the application run slower for no reason since it would not be able to fully utilize the strengths.

In the R Language, every variable is an object, including basic data types like numbers and characters and more complex data types like lists, data frames, matrices, and so on. R, however, does not support purely single-value variables as the data type that supports the basic data types is called an atomic vector, and is considered to be a one-dimensional array. Since every single value variable is technically a vector, operations like these are valid. (*R Language Definition*, n.d.).

```
1 x <- 5
2 y <- c(4, 5)
3
4 print(x * y)
```

```
1 x <- c(1, 2)
2 y <- c(1, 2, 3, 4)
3 z <- c(1, 2, 3, 4, 5, 6, 7, 8)
4
5 print(x * y * z)
```

Figure 5.0 R language atomic vector declaration and operations

Which results in the contents of the largest variable to be modified by the single item variable. This also works for non-single item vectors as long as the item count of each of the vectors is divisible by the item count of the largest vector (See Figure 5.0)

```
[1] 20 25

[Execution complete with exit code 0]

[1] 1 8 9 32 5 24 21 64

[Execution complete with exit code 0]
```

Figure 5.1 R language atomic vector operation results

This kind of variable interaction allows for R to create complex mathematical operations but can be too much for the purposes of tax calculation. If the project was to calculate the taxes not one but many taxes, R would be a perfect programming language. And that will be a lot slower when compared to other languages.

However, R does support something called a Data Frame, which is, in essence, a spreadsheet in R. Compared to other methods of handling spreadsheet-like variables, R's version is rather simplistic and easy to follow. This feature could come in handy as the program makes use of tax tables. However, for a simple application, the feature may not be needed.

```
1 tax_table_2022 <- data.frame(  
2   lower_limit = c(0, 20834, 33333, 66667, 166667, 666667),  
3   upper_limit = c(20833, 33332, 66666, 166666, 666666, Inf),  
4   base_tax = c(0, 0, 2500, 10833.33, 40833.33, 200833.33),  
5   tax_rate = c(0, 0.20, 0.25, 0.30, 0.32, 0.35)  
6 )
```

Figure 5.2 R language Data Frame variable for the 2021 BIR Tax Table

In Go, data types are classified into either basic types or composite types. Go's basic types are the typical types found in many other multi-purpose and multi-paradigm languages, such as Python. The language has these data types: integers, floats, complex numbers, bytes, runes, strings, and boolean expressions (See Figure 6.0).

```

var integer int;
var float float32;
var float2 float64
var string string;
var boolean bool;
var complex complex64;
var complex2 complex128;
var rune rune;
var byte byte;

integer = 1;
float = 1.1;
float2 = 1.2;
string = "string";
boolean = true;
complex = 1 + 2i;
complex2 = 1 + 2i;
rune = 'a';
byte = 'b';

```

Figure 6.0. Basic data types in Go

Composite types in Go are basic or other complex data types that are linked to other data types to create a new data type. Examples of these types include arrays, structs, pointers, functions, interfaces, slices, maps, and channels (Zhang, n.d.).

An array in Go is a numbered sequence of elements that are limited to a single data type. Indexes of an array must always be non-negative (See Figure 6.1).

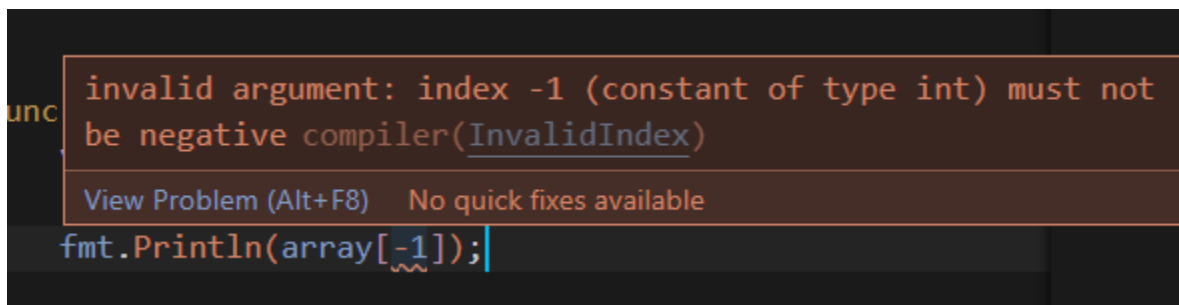


Figure 6.1. Error when trying to access a negative index

Structs are another sequence of elements; however, each element must have a name and type that must be unique.


```

type taxBracket struct {
    nMin      float32
    nMax      float32
    nTax      float32
    percentage float32
}

```

Figure 6.2. Example of a struct used in the tax calculator program

Pointers in Go store the address of a program and can be used to access variables from different functions through a pass-by-reference. Function types denote the set of all functions with the same name parameter and return types. Interface types store methods that must be implemented for a specific struct. Slices are segments of an underlying array and provide access to a specific range of an array. Slices in Go can simulate dynamic arrays by slicing a null array (See Figure 6.3).

```

var array []int;
for i := 0; i < 100; i++ {
    array = append(array, i);
}

```

Figure 6.3. Example of a dynamic array using a slice

Maps are similar to dictionaries or HashTables in other languages as they store key-value pairs. Channels are data types that function like built-in pipes for concurrent functions to communicate with each other.

2.3. Control Structures

Due to the nature of the process of calculating taxes, that being an association of values from a given table, the control structures of a programming language have to be considered when creating the tax calculator. These control structures allow for decision-making, for the purposes of the application domain, associations between cases and values.

Control structures in Kotlin can behave in many different ways. For example, the *if* statement in Kotlin behaves like a function, allowing it to return values. (See Figure 7.0)

```

var sssFee : Float = if(monthlySalary < 0) computeSSS(monthlySalary) else 0f

```

Figure 7.0. Example of a Kotlin *if* statement returning a value

There is additionally the *when* statement. Working similarly to a *switch-case* statement in other programming languages, the *when* statement makes it simple to return values according to a case.

```

fun computePhilHealth(monthlySalary : Float) : Float
{
    return when
    {
        monthlySalary <= 10000f -> 350.00f / 2
        monthlySalary > 1000f && monthlySalary < 60000f -> monthlySalary * 0.035f / 2f
        else -> 2450.00f / 2
    }
}

```

Figure 7.1. Kotlin When Statement

Kotlin also provides standard implementations of looping, such as *for*, *while*, and *do-while* loops for iterative actions. The *for* loop can also be used in different ways, with different corresponding syntaxes. (See Figure 7.2)

```

for (item in collection) println(item)
for (item: Int in ints) {
    // ...
}
for (i in 1 ≤ .. ≤ 3) {
    println(i)
}
for (i in 6 ≥ downTo ≥ 0 step 2) {
    println(i)
}
for (i in array.indices) {
    println(array[i])
}
for ((index, value) in array.withIndex()) {
    println("the element at $index has $value")
}

```

Figure 7.2. Kotlin For Loop Statements

Due to a large number of different control structures in Kotlin, and the different permutations at which they can be used, Kotlin is hindered in simplicity, as a user, when accessing code created by a different user, may not be able to understand the code, due to them learning a different subset of implementations of the control structures.

Most of R's control structures are pretty much the same as their default implementations but with simpler syntax, a new control statement and structure, and some of them working more like functions than structures.

Compared to other languages, R's implementation of the for loop is very simple and helps in both readability and writability (See Figure 8.0). But more importantly there is another feature, which is the **next** statement, which essentially skips the current iteration and moves on to the next (*R Language Definition*, n.d.). (See Figure 8.1)

```
for(i in 1:5) {  
  print(i)  
}  
#1 2 3 4 5  
  
names <- list("Aaron", "Luis", "Austin Powers")  
for(x in names) {  
  print(x)  
}  
#Aaron Luis Austin Powers
```

Figure 8.0. R Language For Loop Statements

```
names <- list("Aaron", "Luis", "Austin Powers")  
for(x in names) {  
  if (x == "Luis") {  
    next  
  }  
  print(x)  
}  
#Aaron Austin Powers
```

Figure 8.1. R Language For Loop Statement using Next Statement

Here are other examples of R's standard control statements

```
x <- 10  
if(x < 5) {  
  print("True~")  
} else {  
  print("False~")  
}
```

Figure 8.2. R Language If Else Statement

```
x <- 5  
while(x > 0) {  
  print(x)  
  x <- x - 1  
}  
#5 4 3 2 1
```

Figure 8.3. R Language While Loop Statement

```
x <- 5
while(x > 0) {
  if (x == 3) {
    next
  }
  print(x)
  x <- x - 1
}
```

#5 4 2 1

Figure 8.4. R Language While Loop Statement using Next Statement

R's implementation of the switch statement makes it so that it acts more like a function than a standard switch statement as you can return values from it which acts similarly to Kotlin's *when* statement. There is also no need to declare a case for case statements and at the same time, the last statement of a switch statement will always be the default value given that it does not have a comparison expression. (See Figure 8.5)

```

x <- "hello"
switch (x,
  "hello" = print("world"),
  "goodbye" = print("see you later")
)
#outputs world

x <- 2
switch (x,
  print("world"),
  print("see you later")
)
#outputs see you later

x <- 2
x <- switch (x,
  "world",
  "see you later"
)
print(x)
#outputs see you later

x <- 2
x <- switch (x,
  x == 2, "world",
  x > 2, "see you later"
)
print(x)
#outputs world

```

Figure 8.5. R Language Switch Statements

R also implements something called a Repeat Loop, which is essentially a while(true) loop, and it can only be stopped by using the break statement. (See Figure 8.6)

```

x <- 10
repeat {
  print(x)
  x <- x - 1
  if(x == 0) {
    break
  }
}

```

Figure 8.6. R Language Repeat Statment

Ruby has support for all statement-level control structures. Selection statements in Ruby include *if* statements and *case* statements, both of which are capable of multiple selections. *If* statements can be followed up by *elsif* and *else* statements (See Figure 9.0) and *case* statements are given a variable and checks the conditions in the *when* statements if it is true for at least one of the conditions (See Figure 9.1). *Case* statements are only preferred over *if* statements when there are many possible conditions and it is checking for only one variable.

```
if income <= 1500
  return contribution
else
  if contribution > 100
    return 100
  else
    return contribution
  end
end
```

Figure 9.0. Ruby If Else Statements

```
case income
when 0...20833
  return 0
when 20834...33332
  return (income - 20833) * 0.2
when 33333...66666
  return (income - 33333) * 0.25 + 2500
when 66667...166666
  return (income - 66667) * 0.3 + 10833.33
when 166667...666666
  return (income - 166667) * 0.32 + 40833.33
when 666667...Float::INFINITY
  return (income - 666667) * 0.35 + 200833.33
else
  return 0
end
```

Figure 9.1. Ruby Case Statements

Iteration control in Ruby includes *for* and *while* loops. *For* loops in Ruby iterates through each element in a collection and *while* loops keep repeating until the condition is false (*Programming Ruby: The Pragmatic Programmer's Guide*, 2001). All the available control structures in Ruby would grant the programmer the ability to express different solutions to a problem.

In Go, there are only a few control structures to emphasize its simplicity. In the program, only a few types of control structures were used, namely the for loop, switch statements, if-else chains, and a break statement used to exit out of a switch case.

2.4. Abstraction Mechanism

Aside from control structures, many features of the discussed programming languages are abstracted away from the user, allowing for code reusability and safety through access modifiers.

For example, Kotlin automatically generates getter and setter functions in classes when not explicitly declared by the user. Aside from that, it implements 4 access modifiers for different levels of accessibility for functions, classes, and variables. That is *public*, which is the default access assignment, *private*, *internal*, and *protected*.

In R, abstraction is primarily done through Functions, Packages, Classes, and Formula Notations. Access modifiers are none existent in R. Package functions are hidden from developers while formula notation allows for users to create complex statistical models or relationships between variables.

In Go, most abstractions are done through the use of functions, interfaces, structs, packages, and channels. These constructs are able to abstract lines of code, especially packages where package functions are hidden from the developer.

Ruby also has abstractions through access modifiers. In a class, a method could be set to either *private*, *protected*, or *public* to define its visibility across the program.

2.5. Overall Strengths and Weaknesses

As discussed earlier, Ruby benefits from it implementing a purely object-oriented paradigm. Instead of these concepts in Ruby being beneficial to the specific development process of the application, it might actually hinder it instead because the weaknesses might affect it more than the strengths in this specific application. The main weakness that would hinder Ruby in this application is that since everything in Ruby is composed of objects, it is inefficient and uses more memory compared to languages that do not use object-based primitive data types. That and coupled with the weaknesses of dynamically typed languages further hinders the efficiency of Ruby during runtime. Its strengths such as flexibility and freedom, expressivity, and the benefits of object-oriented programming are not entirely useful for the development of this specific application.

R is a very powerful programming language when used for complex statistical computations as it is primarily designed for that purpose. Not only that, but R also possesses a powerful graphics system that allows its users to create high-quality graphical visualizations of

data and data plots. R also has a large and active community of users who contribute and support it, and because it is an open-source application, many users may modify it freely allowing for great flexibility. R also has strong data manipulation and handling abilities which make it great for organizing data. And finally, R is easily integrated into other programming languages and tools. That said, it is not without its weaknesses. R is considered to be a very memory-intensive language, which comes hard especially with large datasets and at the same time it makes it slower than other languages without proper and efficient code. R also does not possess a built-in GUI, which does make it not so user-friendly. And finally, R's learning curve is rather steep and can be difficult to learn without a strong background in programming or statistics.

The main strengths of Kotlin are its improvements in Java. Kotlin allows Java code to be run inside Kotlin which makes transitioning between Java and Kotlin programs easy, giving developers time to adapt to Kotlin without starting from scratch. Another advantage is that it makes the tedious process of applying abstraction in object-oriented languages easier by automatically generating getter and setter statements which would make writing code more efficient. It also takes care of the `NullPointerException` present in Java by adding a null safety measure in the language where nullable variables cannot cause a `NullPointerException`. It has a few weaknesses, one of which is regarding efficiency. Since it implements so many features and it is built on top of the Java Virtual Machine, the program runs slower. Another weakness is that Kotlin hinders simplicity as there are many different subsets of features as seen with the control structures.

As a fast and statically typed compiled language, Go is able to compile in a few seconds and efficiently use resources which allows programs to run better, especially in systems with limited resources. Compilation in Go allows high performance as it compiles into native code. The language also has automatic memory management and garbage collection which eliminates overhead. The language was designed to be easy and simple to understand, which allows developers to learn the language quicker, as the language is easy to read and write with. Being a statically typed language, it helps prevent common errors related to data type mismatch, while having better coercion and type inferencing compared to other languages. This is more prevalent when working with composite data types and data structures such as trees and graphs. The language also features a built-in error handling mechanism as well as defer statements similar to finally blocks in other languages to ensure the execution of the program in case an error will occur (See Figure 10.0).

```
func readFile(filename string) ([]byte, error) {  
    f, err := os.Open(filename)  
    if err != nil {  
        return nil, err  
    }  
    defer f.Close()  
}
```

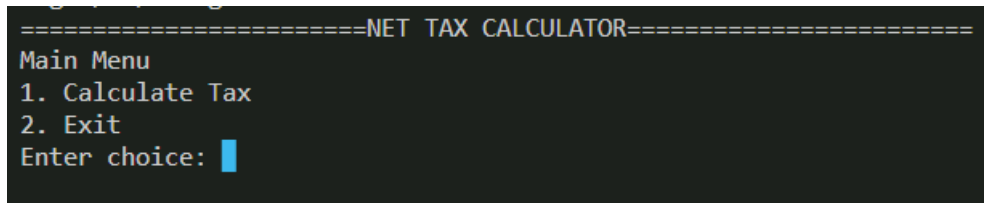
Figure 10.0. Example defer statement

However, the Go language also comes with its limitations and weaknesses. One is the hesitations of developers to add features as they desire to keep the language as simple as possible. An example of this is only the addition of Generics in Go 1.18, approximately 15 years after its development start date. Another weakness is its limited expressibility. Go does not support operator overloading in Go, which could limit the language's ability to write concise code as it would require writing more code just to perform another operation.

3. Developing the Tax Calculator using Go

The tax calculator was developed using Go. Programs in this language are compiled into executable files in the Windows operating system, which allows for accessibility for its users as they would only need this file to obtain the program with all of its uses. This also helps the developers as it is more convenient to instantly compile code into executables, especially for applications such as this one. In the code, many programming features of Go were used to build the tax calculator.

Firstly, the user is greeted with the main menu, wherein the user has to either enter “1” or “2”, depending on whether they want to calculate taxes or exit the application.



```
=====NET TAX CALCULATOR=====
Main Menu
1. Calculate Tax
2. Exit
Enter choice: █
```

Figure 11.0. Main Menu of the tax calculator application

In code, this was constructed using an indefinite for loop, the only looping construct in the language. The for loop in Go does not require the three components usually declared to limit the iterations. The premise of this design choice was to improve readability and writability, as the language will only require the developers to remember one keyword while being able to simulate different kinds of looping constructs, which also further improves the language's orthogonality (Go, n.d.). In the program, an indefinite for loop was used so that the main menu would always appear again until the user decides to exit the application.

```

for {
    fmt.Println("=====NET TAX CALCULATOR=====")
    fmt.Println("Main Menu")
    fmt.Println("1. Calculate Tax")
    fmt.Println("2. Exit")
    fmt.Printf("Enter choice: ")
    var nChoice int
    fmt.Scanln(&nChoice)
    switch nChoice {
    case 1:
        fmt.Printf("Enter Monthly Income: ")
        var nMonthlyIncome float32
        fmt.Scanln(&nMonthlyIncome)

        calculate(nMonthlyIncome)
    case 2:
        os.Exit(0)
    default:
        fmt.Println("Invalid choice")
    }
}

```

Figure 11.1. Code of text-based interface for Man menu

Inside the loop, the user will either be asked to select whether they would like to either calculate tax or exit the program. This is done by using the `Scanln` function provided by the `fmt` package included in the language's standard library which provides formatted I/O. Using the `Scanln` function, the user input is then assigned to a variable previously declared with an integer data type. This variable, "nChoice", will be important to how the program would determine the action to be done, by using a switch statement to control the flow of the program depending on the user's choice. Entering "1" would lead to a code block that would then ask the user to input a number that represents a monthly income. This input would then be assigned to a new variable, "nMonthlyIncome". This variable would then become a pass-by-value argument to a subprogram called "calculate", which is the subprogram that handles all the calculations of the taxes.

The calculated subprogram is in charge of initializing the channels, calling the goroutines, and receiving the results of each goroutine from the channels. These results are then printed into the text-based interface for the user to read. After this subprogram is called, the user is brought back to the main menu, and this loop goes indefinitely until the user decides to exit the program by either closing the executable window or entering "2". To calculate the different taxes and contributions, each type is separated into goroutines that calculate each of the taxes. The program has 4 subprograms in total dedicated to calculating each of these contributions to the net total monthly pay (Refer to Figure 11.2):

```

//call methods to calculate monthly deductions
var incomeTax = make(chan float32)
var sss = make(chan float32)
var philHealth = make(chan float32)
var pagIbig = make(chan float32)

//call the goroutines to calculate the deductions
go calculateMonthlyIncomeTax(nMonthlyIncome, incomeTax)
go calculateMonthlySSSTax(nMonthlyIncome, sss)
go calculateMonthlyPhilHealth(nMonthlyIncome, philHealth)
go calculateMonthlyPagIbig(nMonthlyIncome, pagIbig)

//receive the results
nMonthlyTax := <-incomeTax
nMonthlySSS := <-sss
nMonthlyPhilHealth := <-philHealth
nMonthlyPagIbig := <-pagIbig

```

Figure 11.2. Code of calculate subprogram involving calling goroutines

These subprograms use different features provided by the language, but all of these subprograms require a pass-by-value of the monthly income provided by the user, as well as the channel data object to which the subprogram would send the data.

To calculate the monthly income tax, the calculateMonthlyIncomeTax subprogram was constructed. This subprogram uses a compound type, a user-defined structure to create a type to store tax brackets used by the tax calculator (See Figure 11.3). This struct is created at the beginning of the program, and its first use is in this subprogram. The subprogram creates a slice of taxBracket struct types, to store an indefinite amount of tax brackets (See Figure 11.4). The usage of slice allows developers to change the tax brackets much, if ever these change and new brackets are added or removed, compared to an array with a static number of elements. While slices could have significant effects on runtime performance, the use of slices to simulate a dynamic array does not have many impacts as only operations using slices would have trouble as these are usually expensive and prone to memory leaks (Haryanto, 2021).

```

type taxBracket struct {
    nMin      float32
    nMax      float32
    nTax      float32
    percentage float32
}

```

Figure 11.3. TaxBracket struct

```

var taxBrackets = []taxBracket{
    {0, 20833, 0, 0},
    {20833, 33332, 0, 0.20},
    {33332, 66666, 2500, 0.25},
    {66666, 166666, 10833.33, 0.30},
    {166666, 666666, 40833.33, 0.32},
    //to infinity
    {666666, float32(math.Inf(1)), 200833.33, 0.35},
}

```

Figure 11.4. Slice of taxBracket structs to simulate a dynamic array data type

The subprogram uses a for loop that simulates another kind of looping structure, which is the for-range loop that is generally used to iterate arrays and slices (Educative, n.d.). In the subprogram, this is used to determine the tax bracket range of the given monthly income by using a simple linear search algorithm and sending the resulting tax bracket to the incomeTax channel for the calculate subprogram to receive (See Figure 11.5).

```

var nTax float32
for _, taxBracket := range taxBrackets {
    //finds the first bracket that the monthly income is within
    if nMonthlyIncome >= taxBracket.nMin && nMonthlyIncome <= taxBracket.nMax {
        //return the tax amount for that bracket
        nTax = taxBracket.nTax + (nMonthlyIncome-taxBracket.nMin)*taxBracket.percentage
        result <- nTax
    }
}
result <- nTax

```

Figure 11.5. For-range loop to determine the tax bracket of a given monthly income

The next subprogram that is involved in the calculations of the monthly reductions is the “calculateMonthlySSSTax”, which solves for the amount deducted by the system towards their SSS contribution. This subprogram mainly consists of a variation of a switch statement, called a switch-case statement, which is also equivalent to “switch true”, whereas it evaluates each case inside the statement with different expressions that would result in a boolean value. In this subprogram, the switch case evaluates the cases from top to bottom until it finds the first true expression. Inside each case assigns the value of the SSS contribution to the variable x, where it would then be sent to the “sss” channel (See Figure 11.6). As seen in Figure #, the switch statement of Go does not need to use a “break” control structure, as it would automatically exit the statement once all of the code inside the case block is executed. The use of a switch case was preferred over an if-else chain as it was determined to be simpler and more readable.

```

func calculateMonthlySSSTax(nMonthlyIncome float32, result chan<- float32) {
    //uses a switch case for all ranges of the SSS tax brackets and returns the tax amount
    var x float32
    switch {
    case nMonthlyIncome <= 3249.99 && nMonthlyIncome >= 1000:
        x = 135
    case nMonthlyIncome <= 3749.99 && nMonthlyIncome > 3250.00:
        x = 157.50
    case nMonthlyIncome <= 4249.99 && nMonthlyIncome > 3750.00:
        x = 180
    case nMonthlyIncome <= 4749.99 && nMonthlyIncome > 4250.00:
        x = 202.50
    case nMonthlyIncome <= 5249.99 && nMonthlyIncome > 4750.00:
        x = 225
    case nMonthlyIncome <= 5749.99 && nMonthlyIncome > 5250.00:
        x = 247.50
    case nMonthlyIncome <= 6249.99 && nMonthlyIncome > 5750.00:
        x = 270
    case nMonthlyIncome <= 6749.99 && nMonthlyIncome > 6250.00:
        x = 292.50
    }
}

```

Figure 11.6. Subprogram to calculate monthly SSS tax

The “calculateMonthlyPhilHealth” is a relatively simpler subprogram to create as it did not involve different brackets and other Go features, except that the channel send operation “<-” allows an equation instead of just a variable, which aided in terms of writability (See Figure 11.7).

```

func calculateMonthlyPhilHealth(nMonthlyIncome float32, result chan<- float32) {
    //2022 premium rate
    result <- (nMonthlyIncome * 0.04) / 2
}

```

Figure 11.7. Subprogram to calculate monthly Philhealth Contribution

Lastly, the calculateMonthlyPagIbig subprogram is in charge of calculating the deductions made for Pag Ibig contributions. This uses a walrus operator, a feature in Go wherein the data type doesn’t have to be explicitly typed, compared to a usual variable initialization with a syntax that explicitly declares the data type (See Figure 11.8). However being a statically typed language, the Go compiler has a type inference function that would determine the variable’s data type. This aids in readability as it makes the declaration and assignment of variables and data simpler and more concise.

The contribution is calculated using if-else statements that determine the contribution cost depending on the monthly income and contribution. The result of this is sent to the pagIbig channel.

```

func calculateMonthlyPagIbig(nMonthlyIncome float32, result chan<- float32) {
    contribution := nMonthlyIncome * 0.02
    if nMonthlyIncome <= 1500 {
        result <- contribution
    } else {
        if contribution > 100 {
            result <- 100
        } else {
            result <- contribution
        }
    }
}

```

Figure 11.8. Subprogram for calculating monthly PagIbig contribution.

All of these subprograms are used to calculate the breakdown of the total monthly deductions to a given monthly salary. These subprograms showed different features of Go's data types, control and iterative flow structures, and operators. However, the feature that made the language stand out for this specific task is its rich and native concurrency support, through the use of goroutines and channels.

Concurrency is the ability of a language to perform different operations at the same time. This concept allows programmers and programs to utilize multi-core processing which allows for faster and more efficient programs. A goroutine is a function that is executed independently from the function that called it, meaning that it could run parallel with other goroutines and in a different thread (*Concurrency in GO*, n.d.). Goroutines are lightweight and managed by Go runtime (*A Tour of Go*, n.d.-c). A channel data object can be described as a mediation between goroutines, as it allows them to synchronize without explicit locks (*A Tour of Go*, n.d.-b).

Since concurrency support in the language is native, it is much easier to implement them and would require less time in programming. In Go, goroutines are easy to implement as these are just function calls with the keyword "go" at the beginning of the syntax. Initializing channels in Go is also easy as they are a construct initialized similarly to variables (See Figure 11.9).

```

var monthlyDeduction = make(chan float32)

```

Figure 11.9. Example of channel initialization

The ability to Go to create lightweight goroutines and utilize other concurrency features would improve the program's ability to efficiently calculate all of the tax deductions, as these calculations were programmed to run concurrently with each other.

The program is a Philippine Tax Calculator, and with the use of concurrency, the program would be able to calculate large amounts of data. This ability would flourish when the program is modified to take in a large amount of salary data, and it is tasked to compute the tax distribution of each salary as the workload can be efficiently distributed to multiple CPU cores, allowing the parallel computations of the different monthly deductions.

4. Conclusion

In conclusion, the project was an enlightening experience when it comes to the considerations that need to be taken upon choosing a programming language. For the tax calculation domain, we have extensively explored Ruby, R, Kotlin, and Go. Each of these programming languages has its own strengths and weaknesses as stated before, but the choice of language ultimately depended on the requirements of the project.

Ruby is a dynamic, object-oriented language that is known for its simplicity and ease of use. It is a great choice for the rapid development of simple scripts and web applications. However, since it is mainly interpreted, it is not very user-friendly to those who do not know how to run Ruby code, limiting the accessibility of the program. Additionally, since it is purely object-oriented, it resulted in slow runtimes.

R, on the other hand, is a language designed specifically for data analysis and statistical computing. It has a rich set of libraries and tools for data modeling, visualization, and analysis. And it is more suited for complex tasks that require data analysis, and being able to generate reports with excellent visualization. However, its slow calculation times and steep learning curve did not make it favorable for the application domain.

Kotlin is a statically typed, object-oriented language that runs on the JVM and it is known for its concise syntax, null safety and interoperability with its virtual machine, Java. Kotlin would have been a great option if it came to creating performance-critical applications, but its complexity in features hindered rapid updates, and its slow runtime did not make it favorable for the tax calculation.

Just like Kotlin, Go is a statically typed and compiled language that is known for its simplicity and concurrency support, and excellent compilation times. Excellent for constructing fast performing web applications, network servers, and other systems.

Each of these programming languages has their own strengths and weaknesses, the choice of language will heavily depend on the specific needs and requirements of the project. Ruby would be a good choice for scripts and web applications, R for complex data visualization and analysis, Kotlin for excellent-performing applications, and Go for high-performance web applications and speed.

Ultimately, we decided on using Go language as it is simpler and can be compiled into an executable program, allowing it to be accessible on any computer. Additionally, its simplicity allows for the rapid development and update of the program whenever rules for tax calculations are updated by the government. All in all, Go provided the most features needed for the application domain.

5. References

- A Tour of Go*. (n.d.-a). <https://go.dev/tour/methods/1>. (April 12, 2023).
- A Tour of Go*. (n.d.-b). <https://go.dev/tour/concurrency/2>. (April 12, 2023).
- A Tour of Go*. (n.d.-c). <https://go.dev/tour/concurrency/1>. (April 12, 2023).
- Abhilash, A. (2023, March 27). Operator method call in Ruby. *LinkedIn*.
<https://www.linkedin.com/pulse/operator-method-call-ruby-abhilash-m-a>. (April 12, 2023).
- About Ruby*. (n.d.). <https://www.ruby-lang.org/en/about/>. (April 12, 2023).
- Belov, R. (2017, March 1). *Kotlin 1.1 Released With JavaScript Support, Coroutines, and More | The Kotlin Blog*. The JetBrains Blog.
<https://blog.jetbrains.com/kotlin/2017/03/kotlin-1-1/>. (April 12, 2023).
- Concurrency in GO*. (n.d.). <https://www.golangprograms.com/go-language/concurrency.html>. (April 12, 2023).
- Educative. (n.d.). *Reasons to Love Kotlin - The Ultimate Guide to Kotlin Programming*.
Educative: Interactive Courses for Software Developers.
<https://www.educative.io/courses/ultimate-guide-programming-in-kotlin/>. (April 12, 2023).
- Educative. (n.d.). What is the for-range loop in Golang? [Blog post].
<https://www.educative.io/answers/what-is-the-for-range-loop-in-golang>. (April 12, 2023).
- GeeksforGeeks. (2022). Functional Programming Paradigm. *GeeksforGeeks*.
<https://www.geeksforgeeks.org/functional-programming-paradigm/>. (April 12, 2023).
- GeeksforGeeks. (2023). Kotlin Programming Language. *GeeksforGeeks*.
<https://www.geeksforgeeks.org/kotlin-programming-language/>. (April 12, 2023).
- Go. (n.d.). A tour of Go: Flow control. Retrieved April 12, 2023, from
<https://go.dev/tour/flowcontrol/1>. (April 12, 2023).

Go Fuzzing - The Go Programming Language. (n.d.). <https://go.dev/security/fuzz/>. (April 12, 2023).

Haryanto, A. (2021, April 15). Serious Impact: Slice in Golang for Speed and Memory Usage. Medium.
<https://medium.com/@antoharyanto/serious-impact-slice-in-golang-for-speed-and-memory-usage-c0997f44eb76>. (April 12, 2023).

Indeed Editorial Team. (2022). What is procedural programming language? (And its uses). *Indeed Career Guide*.
<https://uk.indeed.com/career-advice/career-development/procedural-programming-language>. (April 12, 2023).

Ionos. (2020, November 2). *Imperative programming: Overview of the oldest programming paradigm*. IONOS Digital Guide.
<https://www.ionos.com/digitalguide/websites/web-development/imperative-programming>. (April 12, 2023).

JetBrains. (n.d.). *What's new in Kotlin 1.7.0 | Kotlin*. Kotlin Help.
<https://kotlinlang.org/docs/whatsnew17.html#compatibility-guide-for-kotlin-1-7-0>. (April 12, 2023).

JetBrains. (2023a, April 11). *Kotlin roadmap | Kotlin*. Kotlin Help.
<https://kotlinlang.org/docs/roadmap.html>. (April 12, 2023).

JetBrains. (2023b, April 11). *What's new in Kotlin 1.8.0 | Kotlin*. Kotlin Help.
<https://kotlinlang.org/docs/whatsnew18.html>. (April 12, 2023).

JetBrains. (2023c, April 12). *Basic types | Kotlin*. Kotlin Help.
<https://kotlinlang.org/docs/basic-types.html>. (April 12, 2023).

JetBrains. (2023d, April 12). *Numbers | Kotlin*. Kotlin Help.
<https://kotlinlang.org/docs/numbers.html>. (April 12, 2023).

Kotlin Programming Language. (n.d.). Kotlin.
<https://kotlinlang.org/education/why-teach-kotlin.html>. (April 12, 2023).

Kumari, Y. (2023, March 26). *Code Studio*. Coding Ninjas.
<https://www.codingninjas.com/codestudio/library/what-are-the-features-of-object-oriented-programming>. (April 12, 2023).

Lardinois, F. (2019, May 7). *Kotlin is now Google's preferred language for Android app development*.
<https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/>. (April 12, 2023).

- Makala, A. (2021). Golang for Object Oriented People. *DEV Community*.
<https://dev.to/makalaaneesh/golang-for-object-oriented-people-17h>. (April 12, 2023).
- Paramitha, A. P. (2019, December 17). The Benefits of Using the Go Programming Language (AKA Golang). *Mitrais*.
<https://www.mitrais.com/news-updates/the-benefits-of-using-the-go-programming-language-aka-golang/>. (April 12, 2023).
- Pike, R. (n.d.). *Go at Google: Language Design in the Service of Software Engineering - The Go Programming Language*. https://go.dev/talks/2012/splash.article#TOC_1. (April 12, 2023).
- Programming Ruby: The Pragmatic Programmer's Guide*. (2001).
<https://ruby-doc.org/docs/ruby-doc-bundle/ProgrammingRuby/book/language.html>. (April 12, 2023).
- R Language Definition*. (n.d.-a).
<https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Control-structures>. (April 12, 2023).
- R: What is R?* (n.d.). <https://www.r-project.org/about.html>. (April 12, 2023).
- Rodrigues, B. (2022, October 24). *Chapter 8 Functional programming | Modern R with the tidyverse*. <http://modern-rstats.eu/functional-programming.html>. (April 12, 2023).
- Ruby Community*. (n.d.). <https://www.ruby-lang.org/en/community/>. (April 12, 2023).
- Rybacka, K. (2021). The Go programming language - everything you should know - CodiLime. *CodiLime*. <https://codilime.com/blog/what-is-go-language/>. (April 12, 2023).
- Shafirov, M. (2017, May 17). *Kotlin on Android. Now official | The Kotlin Blog*. The JetBrains Blog. <https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/>. (April 12, 2023).
- The Go Blog - The Go Programming Language*. (n.d.). <https://go.dev/blog/>. (April 12, 2023).
- The Go Team. (2022, August 2). *Go 1.19 is released! - The Go Programming Language*.
<https://go.dev/blog/go1.19>. (April 12, 2023).
- What is R? MRAN*. (n.d.). <https://mran.microsoft.com/documents/what-is-r>. (April 12, 2023).
- What's new in Kotlin 1.8.20 | Kotlin*. (2023, April 11). Kotlin Help.
<https://kotlinlang.org/docs/whatsnew1820.html>. (April 12, 2023).

Wickham, H. (n.d.). *Functional programming · Advanced R*.
<http://adv-r.had.co.nz/Functional-programming.html>. (April 12, 2023).

Zhang, K. (n.d.). *The paradigms | The Go Programming Language Report*.
<https://kuree.gitbooks.io/the-go-programming-language-report/content/2/text.html>. (April 12, 2023).

Zharova, E. (2021, February 3). *The state of Go | The GoLand Blog*. The JetBrains Blog.
<https://blog.jetbrains.com/go/2021/02/03/the-state-of-go/>. (April 12, 2023).