Go Programming Language Term Paper

by
Fabian Kirberg

CIS 343
Professor Kurmas
Fall 2022
Grand Valley State University

The Go programming language, or Golang as it is often referred to as a result of the website domain go.org being taken at its creation, began development in September, 2007 at Google, released as an open source project in November, 2009, and had its 1.0 release in March, 2012. Three engineers by the names of Robert Griesemer, Rob Pike, and Ken Thompson decided to create a programming language that was meant to have an emphasis on efficiency as well as combining the aspects of programming languages they felt were beneficial, such as the syntax and readability from languages such as C and Python while trimming out what they thought were unnecessary overcomplexities from C++, which was the language they were at the time primarily working with at Google. Katarzyna Rybacka describes this in her history of the Go language: “Robert Griesemer, Rob Pike, and Ken Thompson grew weary of C++’s complexity, and the lack of a simple language providing efficient compilation and execution” (Rybacka, 2021). The result was Go, an imperative language that supports concurrency and fast compile times. Go is also an object-oriented programming language, however, it is somewhat unique in its implementation and does not provide support for traditional inheritance. Go is used at the company where it was created, Google, but it is increasingly being used by companies and industries where efficiency is required when working with large scalable software systems as well as creating APIs, as Go provides very fast compilation times. As stated in their FAQ, “Go usage is growing worldwide, especially but by no means exclusively in the cloud computing space. A couple of major cloud infrastructure projects written in Go are Docker and Kubernetes, but there are many more” (Go, n.d).

Go data types are divided into four categories which contain a number of subcategories. These data types are divided as such:

1. Basic Types
    a. Numeric (e.g. integers, floats, etc.)
    b. Booleans
    c. Strings
2. Aggregate Types
    a. Arrays
    b. Structs
3. Reference Types
    a. Slices
    b. Channels
    c. Maps
    d. Pointers
    e. Functions

The basic data type category contains the data types that are commonly used across most higher-level programming languages, such as numeric values, bools, and strings. The aggregate data type category contains arrays and structs. Arrays are used in a similar fashion to other languages, where the data is placed next to each other in memory and is accessed by using an index. Structs are used to combine data types together and are used when implementing object-oriented programming with Go, as Go does not use classes. Reference data types are used to refer to where data is stored in memory. Some of the forms of reference data types that Go implements, such as pointers and maps are similar to other programming languages where pointers are used to reference the memory location of data and maps are Go's equivalent of hashmaps or dictionaries. Go categorizes functions as reference data types because Go allows functions to be passed as arguments. Two unique reference data types in Go are Slices and Channels. Slices are address references to a specific subset of an array, while also providing support for determining the length and capacity of the slice. Channels are

used in combination with goroutines, which are lightweight threads that execute concurrently with the rest of the program, the benefits of which are highlighted by Yaroslav Bai in his support for Go, “One of the reasons why you ought to use Golang is its ability to support concurrency. The Go language has Goroutines, which are basically functions that can run simultaneously and independently. (Bai, n.d)” Channels are used to send or receive goroutines, which allows for a program to account for synchronization with respect to waiting for the completion of goroutines that may be running concurrently.

Go is a statically typed language, and as a result type checking occurs at compile time. Variables can be either explicitly or implicitly typed in Go, this is done using either the `var` declaration when doing so explicitly or the `:=` operator when doing so implicitly. An important distinction to be made is that in Go, the data type is written after the variable name.

```
//These achieve the same result
var num int = 5
num := 5
```

While many statically typed languages provide implicit type conversion between compatible data types, this is not the case for Go. All operations or assignments between two variables of different data types will require the explicit type casting or conversion of one variable to the other, or else there will be a compile-time error. The benefit of being statically typed and requiring explicit type casting for conversions is that Go provides a high level of reliability in its code, however, this comes at the cost of have less writability.

Go is based in large part on languages such as C++ and Java, it borrows much

of the syntax used by these languages. However, Go also has many properties that make it unique and while knowing C++ or Java will help one more quickly understand Go, it is important to know its distinctions.

The hierarchy of operator precedence in Go is as follows:

Priority	**Operators**
1	* / % << >> & &^
2	+ - | ^
3	== != < <= > >=
4	&&
5	| |

Binary operators that are of the same precedence will follow left-to-right associativity.

Selection constructs in Go are very similar to those found in C, with a few but major differences as well as some minor differences. The minor differences are that in Go, the syntax does not include parentheses around the conditions for loops. The major differences are that in Go there are no `while` or `do` loops, which are instead implemented using the `for` loop keyword as well. While this may provide for better writability, the tradeoff is that it also makes for less readable code. For example, here is a `while` loop used in my Connect 4 project for C to check for a win in the down-right-diagonal direction and the same “while” loop used in Go:

```
// In C
while(/*Check Conditions*/){
      down_right_counter++;
}
// In Go
for /*Check Conditions*/ {
      down_right_counter++;
}
```

Switch statements in Go evaluate the potential matches for a switch case from top to bottom and also allow for expressions other than integers, similar to `if-else` chains.

Iteration in Go is also similar to other popular languages. To iterate through arrays or strings in Go, one uses a `for` loop. This can be done like in C, where an incrementer is declared and used to iterate the length of the array. One can also use the `range` keyword to iterate through an array as well.

```
array := []int{1,2,3}
// Standard Incrementer
for i := 0; i < len(array); i++ {
    fmt.Println(array[i])
}
// In range incrementer
for index, array := range arr {
    fmt.Println(array[i])
}
```

Functions in Go are used in a similar manner to C but with come with some useful additions to their functionality. In Go, functions are able to return multiple values. This can be used to utilize only one function to return two or more related values, instead of having to use multiple functions. This provides a significant increase in writabilty to Go over other languages.

Scope rules in Go are also similar to C. In Go global variables are variables that are declared outside of all functions and can be accessed anywhere in the program. Local variables are declared inside a function or a block of code and can only be accessed while within that function or block of code. A function's parameters are treated as local variables within itself and shadow any outside variables.

Importing packages or modules into Go is done using the import keyword. For example: `import "fmt"` or `import "[File Name]"`. Packages usually have short

and distinct names to make using a package and its contents easy to do, however, if required, an imported package can be set to use a different name for accessing its contents.

Exception handling in Go is done using built-in error types. For example, if calling a function, immediately afterward, the returned error type can be checked to see if it is not null. If this is the case, an error has occurred and the resulting code block is used to handle the error. This form of error handling provides an increased level of reliability to Go. For example, my implementation of parsing user input for Connect 4 in Go, which can be seen in Appendix A, parses the number of rows and columns to be used in the creation of the game board. However, if it is unable to parse the input to an integer, the return error value will not be null, meaning an error has occurred and must be handled or the program will exit.

Go provides support for object-oriented programming in a unique manner when compared to the languages it stems from, such as C++ and Java. Go does not support classes but instead only structs and as a result, Go also does not provide support for traditional inheritance, polymorphism, or interfaces, although something similar can be achieved by embedding parent structs in child structs. This question of whether Go is an object-oriented language is answered on the language's official FAQ: "Yes and no. Although Go has types and methods and allows an object-oriented style of programming, there is no type hierarchy. The concept of "interface" in Go provides a different approach that we believe is easy to use and in some ways more general. There are also ways to embed types in other types to provide something analogous—but not identical—to subclassing" (Go, n.d). To make variables and

functions in a struct public their names must be capitalized, if a variable or function name is uncapitalized it means that they are private and can not be accessed outside of the struct. By not supporting classes and using this unusual form of encapsulation, Go reduces it writability and readability. This encapsulation functions similarly on a greater level in regards to packages, as a struct or a method can only be accessed in a different package if they are made public via capitalization. For an example of this design, see how I utilized Go's object-oriented approach in my implementation of Connect 4 in Appendix B.

Overall, Go is a language that is similar to C++ and Java but provides some extra benefits in terms of writability, however, these benefits potentially lower its readability. Being able to return multiple values with one function in Go is something that can be used to greatly reduce redundancy and increase the compactness of one's code, however, the downside is that it can very easily lower the readability of code and often also goes against the premise of reducing code into smaller chunks. Having variables and functions be public or private based on whether they are capitalized or not is also a very minor increase in writability for a significant decrease in readability. Go has a high amount of reliability due to being statically typed as well as implementing built-in error return values for its functions, which in turn also provide for better writability and readability than try/catch blocks. Being statically typed alongside features such as using Channels alongside goroutines makes Go a very orthogonal language, ensuring that every part of the code runs as intended. Go is a good choice for implementing Connect 4 because it allows for easier error handling and input validation while being similar to the well-known programming languages it is in part based on like C, C++, and Java.

However, the cons of implementing Connect 4 in Go outweigh the pros. I believe that this is in large part due to the fact that Go is intended for very large software projects and the benefits are not as apparent on a project of this scale, especially considering that one of the most emphasized aspects of Go is its fast compile time for being statically typed, which for a Connect 4 project is not noticeable. These points make Go a valuable language to learn if working on large-scale projects due to its efficiency, reliability, and easy-to-learn nature as it extends languages most programmers likely already know.

Works Cited

Rybacka, K. (2021, September 17). *The go programming language - everything you should know*. CodiLime. Retrieved November 26, 2022, from https://codilime.com/blog/what-is-go-language/

Chris, K. (2021, October 7). *What is go? Golang programming language meaning explained*. freeCodeCamp.org. Retrieved November 26, 2022, from https://www.freecodecamp.org/news/what-is-go-programming-language/

Bai, Y. (n.d.). *Best practices: Why use golang for your project*. Best practices: Why use Golang for your project. Retrieved November 26, 2022, from https://www.uptech.team/blog/why-use-golang-for-your-project

GolangByExample. (2022, July 14). All data types in Golang with examples. Retrieved November 26, 2022, from https://golangbyexample.com/all-data-types-in-golang-with-examples/#Reference_Types

Go. (n.d.). A tour of go. Retrieved November 26, 2022, from https://go.dev/tour/welcome/1

Appendix A

```
if match {
      var splitString = strings.Split(os.Args[1], "x")
      if rowNum, err := strconv.Atoi(splitString[0]); err == nil {
            rows = rowNum
      }

      if colNum, err := strconv.Atoi(splitString[1]); err == nil {
            cols = colNum
      }
} else {
      fmt.Println("Board size " + os.Args[1] + " is not formatted
properly.")
      os.Exit(0)
}
```

Appendix B

```
type Connect4 struct {
    num_rows    int
    num_columns int
    win_length  int
    totalSlots  int
    board       []int
}

// Constructor
func New(num_rows int, num_columns int, win_length int) *Connect4 {
    c4 := new(Connect4)
    c4.num_rows = num_rows
    c4.num_columns = num_columns
    c4.win_length = win_length
    c4.totalSlots = num_rows * num_columns
    c4.board = make([]int, c4.totalSlots)
    return c4
}

// Helper method to print column labels
func (c4 Connect4) Header() {
    fmt.Println(" A B C D E F G H I J K L M N O P"[0 : c4.num_columns*2])
}
```