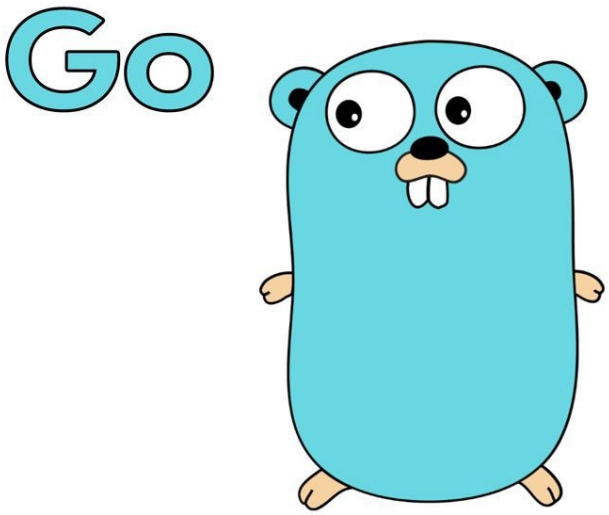
Go! – An Analysis And Review



"The designers were primarily motivated by their shared dislike of C++"
-Dr. Dobbs, 2011

1. Introduction

The year was 2010, and the nation was collecting its wits after being blindsided by the "California Girls" music video released by pop culture hit phenomenon Katy Perry. It was this fateful year that google engineer Rob Pike found himself asking a question that was lingering in the minds of many developers at the time: "Do we have enough programming languages?"[1]. Indeed, prior years had seen a variety of languages of exceedingly different styles and syntax attain widespread use, such as Python, Ruby, Scala, Erlang, and Lua. While these languages each offer unique features that can be leveraged effectively in a variety of situations, Mr. Pike and his peers found themselves unsatisfied and proceeded to then aptly answer their own question with the **Go programming language**, which was conceived to address design challenges being faced by Google in the early 2010s. The designers wanted a language that[3]:

- Maintained static typing systems in C-based languages
- Had good readability and utility as found in languages such as Python and Javascript
- Perhaps most prominently, could accommodate the design of high performance networking and multi-processing applications.

With these design objectives in mind, Go was thusly birthed from the collective minds of Robert Griesemer, Rob Pike, and Ken Thompson.[2] Fast-forward to the modern era, and Go has found widespread use and adoption in major software projects across the entire spectrum of the tech industry. Behemoths such as Netflix, Dropbox, Cloudflare, Ethereum and many, *many* other projects have employed the unique features of Go. The language lends itself especially well to concurrent design which allows for exploitation of parallel hardware and helps to deliver a subsequently better experience to the end user. The need for a language that allowed for the *productive* design of complex back-end computing solutions was clear, and Go has shown itself to be a particularly well suited to these tasks.

Go has a wide feature set which will be explored in detail in this paper. Currently, it is in its first iteration of version 1.0, but is under draft for a version 2.0. We will review and analyze the feature set present in version 1.0. Version 2.0 is largely the same, but introduces support for generics and additionally employs less verbose error handling. There are several reasons why generics were not included in the initial implementation, and we will explore this further in the discussion regarding type systems and subprogram features.

2. Computational Model

Go maintains active residence in the fabled land of C-based languages, and as such retains an imperative syntax. Go has functions which are constructed from statements, and these are parsed as imperative commands. Go is as such a procedural language, as functions are executed in sequence to form a program. It is notably however, multi-paradigm, in that it can be written in a functional or object oriented style as needed to suit the task at hand. It is compiled and thus not a scripting language.

3. Hello World

We realize a simple Hello World application in the Go language, taking note of some key features:

```
// package main tells the compiler to output an executable
//binary instead of a shared library
package main

// import statement brings in the fmt package
// to allow printing to the console
import "fmt"

// main function declared with the func keyword
// initiates program
func main() { fmt.Println("hello world") }
```

fig 1: A simple hello world application

Of particular note is the white space between each line. Ignoring the comments, one line of whitespace is automatically added by the Go linting tool included with the standard Go installation between function declarations, import statements, and package membership statements, and this reflects the desired styling conventions of the developers. Go is compiled, and so in order to to run a Go program, one must first build:

```
dr@thiccpc:~/Documents/School_Related/TCSS/TCSS 380/Paper$ go build hello_world.go
```

And then run:

```
dr@thiccpc:~/Documents/School_Related/TCSS/TCSS 380/Paper$ go run hello_world.go
hello world
```

The avid reader will also note the lack of end line indicators such as ; as the end of line is inferred by the new line character and the use of brackets.

4. Type System

The type system of Go has the common types from the C language family that we know and love.

In code, we provide the following example:

```
package main
import "fmt"

var (
    //Go retains the usual types of the C based languages:
    byte_example uint8 = 255
    sixteen_bits uint16 = 65535
    sixtyfour_bits uint64 = 0xFFFFFFFFFFFFFFFF
    a_boolean bool = false
    a_byte byte = 255
    zero_value int
    rune1 rune = '♬'
)
var no_type = "no problem"
func main() {
    fmt.Printf("Type: %T Value: %v\n", byte_example, byte_example)
    fmt.Printf("Type: %T Value: %v\n", sixteen_bits, sixteen_bits)
    fmt.Printf("Type: %T Value: %v\n", a_boolean, a_boolean)
    fmt.Printf("Type: %T Value: %v\n", sixtyfour_bits, sixtyfour_bits)
    fmt.Printf("bytes are equal to uint8: %v\n", a_byte == byte_example)
    fmt.Printf("no value? initialize to zero: %v\n", zero_value)
    fmt.Printf("Runes display unique unicode values: %v\n", rune1)
    fmt.Printf("Go can also infer types: %T\n", no_type)
}
```

fig 2: various type declarations in Go

Running the above application yields the following output:

```
dr@thiccpc:~/Documents/School_Related/TCSS/TCSS 380/Paper$ go run types.go
Type:  uint8  Value:  255
Type:  uint16  Value:  65535
Type:  bool  Value:  false
Type:  uint64  Value:  18446744073709551615
bytes are equal to uint8: true
no value? initialize to zero: 0
Runes display unique unicode values: 9836
Go can also infer types: string
```

Our example shows the great flexibility of the typing system in Go. Indeed, the designers have created a language which has combined the static typing of the C family of languages with the improved readability of Python style scripting languages. Also, make note of the fact that variables can be declared in groups or individually with the use of the var keyword and parenthesis.

The following common types are available for use and can be employed either explicitly or implicitly:

Type	Value
int8	8-bit signed integer
int16	16-bit signed integer
int32	32-bit signed integer
int64	64-bit signed integer
uint8	8-bit unsigned integer
uint16	16-bit unsigned integer
uint32	32-bit unsigned integer
uint64	64-bit unsigned integer
int	same size as uint, either 32 or 64 bit.
uint	Same size as int, either 32 or 64 bit.
rune	Alias for Unicode.
byte	Alias for int8.
uintptr	Has undefined width but is capable of storing adequate bits to represent a pointer.

4.1 Lack of support for generics

We have already asked ourselves many important and game-changing questions in this paper:

1. How does Katy Perry get away with it?
2. Are there enough programming languages?
3. Am I dehydrated?

While experts agree that the average human adult should consume between 2.7 to 3.7 liters of water per day[10], the scope of this paper remains firmly centered on the history of a programming language designed with the namesake of the humble American common gopher, and thus we remain resolute in this task. Lacking support for a fundamental feature belonging to many common languages leads us to a new quandary: "How could a language that prioritizes productivity and simplicity omit a feature that is so fundamental to efficient production of software?"

While it is true that generics, polymorphism, and other forms of abstraction can add simplicity and efficiency to the software engineering *process*, these features add a great deal of complexity to the underlying design of the language itself. This fact directly contradicts the stated goals of the language designers in Go v1.0, and as such, support for generics

were not included.[9] The Go v2.0 design draft is planned however, to include support for this feature due to desires of the developer community at large.

4.2 Alternatives to generics

When facing a lack of support for a particular design paradigm, a common and effective strategy in software development is to think about the problem differently. And so we press on, aided by the fact that the language contains support for a few generic data structures that again are common to imperative languages. In Go, we can make use of:

1. Arrays
2. Slices (sections of arrays)
3. Maps

All of which will allow us some form of generic functionality. Let's take a look at the map functionality in Go. Maps are an extremely versatile and often efficient strategy for storing collections of "things"; (this advantage is on display prominently in Javascript). In Go we can leverage a map in the following fashion:

```
map[type]{struct}
```

Looks to be generic enough right? We can insert keys of any type that will map to values of any type, including those of a custom variety. In most situations, this might be good enough to solve the problem at hand. In code, we demonstrate the following:

```
func main() {

    //instantiate a new type
    type animalLanguage struct {
    animal string
    }

    //kinda generic and good enough in some situations
    sortOfGeneric := make(map[string]animalLanguage)

    sortOfGeneric["python"] = animalLanguage{animal: "snake"}
    sortOfGeneric["rust"] = animalLanguage{animal: "rustacean"}
    sortOfGeneric["go"] = animalLanguage{animal: "gopher"}

    fmt.Println("map:", sortOfGeneric)

}
Output: map: map[go:{gopher} python:{snake} rust:{rustacean}]
```

fig 3: Learn rust, become rustacean

When does this design approach become cumbersome? In a generic implementation, we would expect a collection of abstract methods that provide functionality such as sorting, comparing, or testing for equality. In Go, we can absolutely sort a map, but it takes extra work that some may find undesirable or obstructionist to their workflow. Let's take a look:

```go
package main

import (
    "fmt"
    "sort"
)

//instantiate a new type
type animalLanguage struct {
    animal string
}

func main() {

    //kinda generic map and good enough in some situations
    sortOfGeneric := make(map[string]animalLanguage)

    sortOfGeneric["python"] = animalLanguage{animal: "snake"}
    sortOfGeneric["rust"] = animalLanguage{animal: "rustacean"}
    sortOfGeneric["go"] = animalLanguage{animal: "gopher"}
    sortByString(sortOfGeneric)
}

func sortByString(sortMe map[string]animalLanguage) {
    keys := make([]string, 0, len(sortMe))
    for k := range sortMe {
        keys = append(keys, k)
    }
    sort.Strings(keys)
    for _, k := range keys {
       fmt.Println(k, sortMe[k])
    }
}
```

fig 3.1 What we had to do was write an entirely new sorting function from scratch, create an extra data structure, and an extra loop. This works, but calling it elegant is generous.

It may not seem entirely inconvenient to add such functionality by hand in this fashion, but time spent performing monotonous tasks such as above contradicts the Go design philosophy of focusing on efficient software development, which may explain why support for generics is planned for future releases.

5. Variables

As we have shown above, it is the case that variables in Go can be either inferred, static, explicit/implicit. We present another example which demonstrates the same concept used in the previous type example. We can use function closures as a way to declare a global static variable that can be reused anywhere in code:

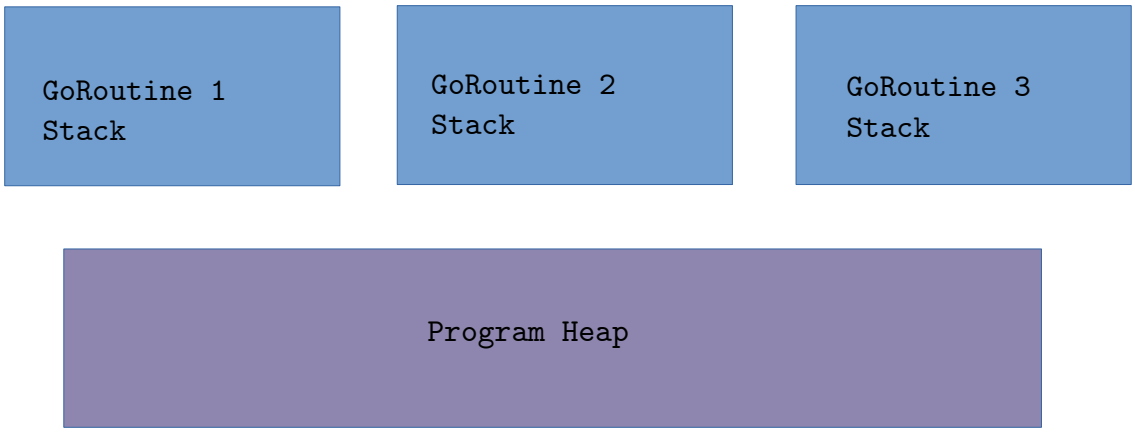
```
package main

var this_is = "a_static_variable "

func main() { println(this_is) }
```

fig 4: variables can be declared in a global context and then preserved as static, since function literals are closures in Go.[8]

Variable placement in memory in Go is dependent on the scope of the function in which the variable is declared. Go introduces the concept of Go routines, which is similar to threading and is useful for abstracting away the memory management aspects of Go. In the following diagram, we show the address space layout of a compiled Go program:



There are a few important questions to answer regarding the design of this model:

1. How does a GoRoutine stack grow as the program executes?
2. Where are variables stored in memory during compilation?
3. How does garbage collection work in this model?

In regards to memory growth, fear not, dear reader, as a GoRoutine stack is initialized with 2kb of memory by default. It can then grow as necessary to accommodate the memory requirements of the given GoRoutine. This implies that stacks in Go are

dynamically sized and can keep growing within the memory limits of the overall hardware in use.

During a build, the compiler will do its best to place variables that are local to functions within the stack frame of the corresponding GoRoutine. However, if the compiler detects that a variable is not referenced following the function return, it will place the variable onto the garbage collected heap, and in this way, the dangers of dangling pointers are mitigated.

We proceed to gain insight into how mutability is approached within the Go programming language. Your cut-and-dry, modern *pure* functional style language will provide you with immutable values, in that once declared, their value cannot be reassigned. Traditional imperative style languages may offer support for either approach depending on the specific language, and even many modern functional style languages offer some form of mutable variables. Go maintains the conventions of the C family of languages in that strings are immutable, but other variable types can be changed in place.

Behold, dear reader, as we daringly attempt the impossible: (*If I don't make it back from this, tell my family I love them and someone please delete my web browser history*)

```
func main() {
    s := "A string"
    fmt.Printf("The first character in s: %v", s[0])
    s[0] = 'Z'
}
$ go build mutability.go
# command-line-arguments
./mutability.go:8:7: cannot assign to s[0]
```

fig 5: tragic failure

Figure 5 shows the immutability of strings in the Go programming language. Other common types however, can be changed in place as needed:

```
func main() {

    x := 5
    x = x + 5
    fmt.Printf("Value of x is: %v\n", x)

}
Output: Value of x is: 10
```

fig 6: Redemption

6. Subprogram Syntax

Go is compiled, and as such requires a bit of set up before lines of code are actually executed. Interpreted REPL languages accept input to a console and can usually be executed in single statements, at the cost of performance however, because even though this high level abstraction allows for quick statement execution, in the words of legendary software design juggernaut professor Charles Bryan from the University of Washington:

"*Really what it's doing is taking your code and winking at you, and then working some magic behind the curtain".*

Such powerful truth and wisdom cannot be understated, but fortunately at a high level, Go lets us write comparatively few lines of code to get down to executable statements.

```cpp
//C++, the motivation for Go
#include <iostream>

using namespace std;

int main(int argc, char * argv[])
{
     cout << "Hello, world!" << endl;
     return 0;
}
```

```go
//Go
package main

import "fmt"

func main() { fmt.Println("Hello, world!") }
```

fig 7: In the game of code golf, Go comes out ahead on par, compared to C++ with a score of 5, sadly over the par for the course. A stunning upset by Go and truly one for the history books.[11]

As we have shown in previous examples, the class structure which is present in Java is not visibly present in Go. Once the package is declared and the appropriate import statements are added, functions can be written stand-alone. Naturally, this may lead us to ponder the nature of inheritance or polymorphism in Go, which as we will show is closer to that of C in the use of structs. We can create a sort of pseudo-polymorphism through the use of *composition,* in which the fields of the struct can form fields for a new struct:

```go
type apple struct{ appleType string }
type orange struct{ orangeType string }

type experimentalFruit struct {
     apple
     orange
}
fruit_basket := experimentalFruit{
     apple{appleType: "cosmic crisp"},
     orange{orangeType: "tangelo"},
}
```

fig 8: composition of structs as a stand-in for polymorphism

7. Subprogram Parameters

We demonstrate how Go is strictly pass-by-value in the following example:

```
type exampleType struct { //instantiate a new type
    example string
}
//modify a field from the type passed in
func (t exampleType) exampleFunc() {
    t.example = "Am I different?"
}
func main() {
    var exampleTest = exampleType{example: "Here I am, here I remain"}
    exampleTest.exampleFunc()
    println(exampleTest.example)
}
```

fig 9: Mystery! Intrigue! Does the exampleFunc change the field value of the passed type?

Running the above results in the following:

```
dr@thiccpc:~/Documents/School_Related/TCSS/TCSS 380/Paper$ go run variables.go
Here I am, here I remain
```

In the above example, we show how a new type is instantiated through the use of a struct. After setting the field value in main, we pass the type to exampleFunc and attempt to modify the field value. This fails however, due to the fact that Go strictly passes parameters by value. A copy of the exampleType variable was made inside of the exampleFunc stack frame, and hence printing the field value in the variable declared in main produced the original variable as it was before exampleFunc was applied.

Go allows usage of pointers and addresses just as expected in most C-based languages:

```
func main() {

        var x int = 100
        var y int = 200

        fmt.Printf("Value of x : %d\n", x)
        fmt.Printf("Value of y : %d\n", y)

        /* &works as expected to store the address of a variable */
        swapByAddress(&x, &y)

        fmt.Printf("After swap, value of x : %d\n", x)
        fmt.Printf("After swap, value of y : %d\n", y)
}
func swapByAddress(a *int, b *int) {
```

```
        var temp int
        temp = *a //save the value at address x
        *a = *b //put y into x
        *b = temp //put temp into y
}
Value of x : 100
Value of y : 200
After swap, value of x : 200
After swap, value of y : 100
```

fig 10: The ol' switcheroo

Go lends itself to effective low level memory access just as we would expect in a C-based language. This allows for quick and effective management of data stored in memory, and maintains the safety and reliability that comes with a pass-by-value design principle.

7.1 Subprogram parameter conventions

Parameters to subprograms in Go are positional, in that there is no inference of a parameter based on the type or how it is passed in. Watch us rudely upset the compiler:

```
func main() { printMe(42, "the meaning of life") }

func printMe(a_number int, a_string string) {
        fmt.Printf("%v is %s\n", a_number, a_string)
}
$ go run subprogram_params.go
42 is the meaning of life
```

fig 11: no problems here

```
func main() {
        printMe("the meaning of life", 42)
}

func printMe(a_number int, a_string string) {
        fmt.Printf("%v is %s\n", a_number, a_string)
}
$ go build subprogram_params.go
# command-line-arguments
./subprogram_params.go:9:10: cannot use "the meaning of life" (type untyped
string) as type int in argument to printMe
./subprogram_params.go:9:33: cannot use 42 (type untyped int) as type string
in argument to printMe
```

fig 11.1: The author apologizes dearly for this brash disregard for the rules.

7.2 Default values and reserved keywords

Go has default values that are automatically assigned to uninitialized variables and parameters by the compiler. In the opinion of the author, they are sensible and work mostly as expected. Let us revisit our beautiful type chart that was constructed earlier, with some additional entries:

Type	Default Value	Type	Default Value
int8	0	float	0
int16	0	complex	$0 + 0i$
int32	0	boolean	false
int64	0	*pointer	nil
uint8	0	struct	Fields take on zero values of types they are composed of
uint16	0	slice	nil, with length and capacity 0
uint32	0	map	nil
uint64	0	interface	nil
int	0		
uint	0		
rune	0		
byte	0		
uintptr	0		

The reserved keywords[12] in Go are the following:

Declaration	Composite types	Control flow		Function Modifier
const	chan	break	goto	defer
var	interface	case	if	go
func	map	continue	range	
type	struct	default	return	
import		else	select	
package		fallthrough	switch	
		for		

The standard naming conventions apply that are found in most languages, and the linting tools provided with a Go installation will assist in making sure that these conventions are followed. (i.e. cannot start with numbers or non-alphabetical characters, etc).

8. Additional Subprogram features:

Earlier, we had dangled the proverbial carrot of functional style features present in the Go language. At long last, it is time for us to explore this feature further. First, a quick review of some terms that are relevant to this topic of discussion[13]:

- First class function: Functions that are treated as variables
- Higher order functions: A function that accepts another function as an argument and/or returns a function
- Anonymous functions: nameless functions defined once and only used in the context in which they are defined.

Go has support for all of these features. Feast your eyes on the following:

```
func sum(x, y int) int {return x + y}
/* contrary to other familiar languages, the
function signature comes after the argument list */
func partialSum(x int) func(int) int {
       return func(y int) int {return sum(x, y)}
}
func main() {
       partial := partialSum(5)
       fmt.Println(partial(2))
}
$ go run function_passing.go
10
```

fig 12: Eat your carrots and you'll grow up to be just like Javascript

The world is confusing, and we came to this paper answers! So let's think about what is going on here. First, we define a "*variable*" which is actually a call to the function `partialSum(5)`. This sets `partial` to a function that will return the `sum` function, which in turn accepts two integers and returns their sum. Following the declaration definition of `partial,` since it is a function, we call it with the value 2, which returns the value 7 as expected. Apparently the world also has too many birds, and we just killed roughly three of them with the same stone. The line in `partialSum` that defines the return value (which again, is itself a function) is an example of an anonymous function, as it has no name and is only used in the context of the function where it is defined.

9. Subprogram return mechanism

We presented the conventions surrounding function signatures in the example above. Let us take another look at the return mechanisms in use in Go.

```
func returnToSender(delivery string) (content string) {
	content = "bird" + delivery
	return
}

func main() {
	payload := returnToSender(" golf")
	fmt.Println(payload)
}
$ go run return_example.go
bird golf
```

fig 13: Nobody ever wins at the game of bird golf.

There is nothing terribly different or shocking about this setup, except for some semantic rearrangement of code. The function signature includes the specific value and type to be returned and appears after the argument list as we have already shown. The biggest difference from other familiar languages is simply the placement of the return value in the signature as opposed to the return statement itself. This may or may not improve readability depending on the developer, as the return value is front and center in the reader's field of vision.

Go supports multiple return and fires back a tuple of whatever is defined in function signature. This simultaneously implies support for multiple assignment as well:

```
func multiple_return(x int, y int) (int, int) {
	return x + y, y * x
}

func main() {
	first, second := 5, 5
	third, fourth := multiple_return(first, second)
	fmt.Printf("third: %v \nfourth: %v\n", third, fourth)
}
$ go run multiple_assignment.go
third: 10
fourth: 25
```

fig 14: multiple assignment works both to initialize values and also to assign values returned from function calls.

10. Other notable features: Concurrency

Go was designed with concurrency in mind. Indeed, concurrency facilitates massive scaling of software-based solutions and has led to a proliferation of an astounding variety of digital products and web-based services in the modern era. This is because in many large-scale distributed software applications, it is generally better to exploit available parallel hardware when it is available. This approach is certainly possible in C and its plethora of derivatives, and it is accomplished in a more straightforward sense in some languages rather than others. We wish to show a simple example borrowed from "Go by example"[14] That shows a quick and easy asynchronous setup utilizing the built-in goroutine construct:

```
func thread_test(from string) {
	for i := 0; i < 3; i++ {
		fmt.Println(from, ":", i)
	}
}

func main() {

	thread_test("direct")
	go thread_test("goroutine")
	go func(msg string) {
		fmt.Println(msg)
	}("(go)" + "ing")
	time.Sleep(time.Second)
	fmt.Println("done")

}
```

```
$ go run concurrent.go
direct : 0
direct : 1
direct : 2
go-ing lol
goroutine : 0
goroutine : 1
goroutine : 2
done
```

```
$ go run concurrent.go
direct : 0
direct : 1
direct : 2
goroutine : 0
goroutine : 1
goroutine : 2
go-ing lol
done
```

fig 15: the run on the left shows what we may expect from a synchronous execution, while on right we clearly see an interleaving between thread calls where our dumb pun appears in a different sequence than the first run.

This is a simple example of threading within the Go language. We defined the concept of goroutines in our earlier exploration of the Go memory model, and we also presented the **go** reserved keyword in section 7.2. The **go** keyword initiates a new asynchronous thread which in turn has its own stack frame in the memory layout. Upon execution, we have a blocking call which is the call to `threadTest(“direct”),` the output of which always appears first. The next two calls initiate the goroutines and are asynchronous, and we show how their outputs may be interleaved depending on how the CPU decides to schedule them. What is somewhat striking about this result is the relative simplicity in which we were able to get this up and running.

We find the above to a particularly exemplary instance of Go code that represents the design principles of Go that we presented in the beginning of this research. With relatively few lines, we are able to realize a multi-threaded application which is robust and and took little time to implement, which, (to be frank) is a non-trivial task in a language such as C/C++ and even Java.

Conclusion

It turns out that back in 2010, we did in fact not have enough programming languages. Through our exploration of the Go language that we have carried out, it seems safe to draw the conclusion that Mr. Pike et. al have successfully realized their vision of a language that is highly concurrent, readable, and relatively easy to implement and maintain, which may expound the great number of entities within the tech industry who employ the use of Go today. The C language family has many merits, including C itself, but it is the opinion of this author/developer that larger movement towards language design that safely provides high level abstractions in regards to non-trivial constructs in software is of great use towards the design of robust, useful, and high performance software.

Our exposure to Go prior to this work was minimal to non-existent, (outside of a few cursory glances on github), however in a relatively short period of time we became comfortable enough to write a thorough dossier on the finer aspects of the language, and so hopefully this work speaks for itself in regards to the accessibility of Go. We did not have enough time or space to cover every detail here, but we conclude this paper in the hopes that our audience has enjoyed reading it as much as we enjoyed writing it. ■

Sources:

[1] “Abstract/Bio.” Stanford EE Computer Systems Colloquium, web.stanford.edu/class/ee380/Abstracts/100428.html. Accessed 3 Mar. 2022.

[2] "Language Design FAQ". golang.org. January 16, 2010. Retrieved February 27, 2010.

[3] "Frequently Asked Questions (FAQ) - The Go Programming Language". golang.org. Retrieved February 26, 2016.

[4] "The Go Blog: Go 2 Draft Designs". August 28, 2018.

[5] Novi, Pipit. “The Benefits of Using the Go Programming Language (AKA Golang).” Mitrais, 17 Dec. 2019, www.mitrais.com/news-updates/the-benefits-of-using-the-go-programming-language-aka-golang/#:%7E:text=Go%20is%20a%20procedural%20programming,generally%20phrased%20as%20imperative%20commands.

[6] “Go Type System Overview -Go 101.” Go Type System Overview -Go 101, go101.org/article/type-system-overview.html. Accessed 4 Mar. 2022.

[7] Kirk, James. “Understanding Allocations in Go: The Stack, the Heap, Allocs/Op, Trace, and More | Eureka Engineering.” Medium, 24 Dec. 2021, medium.com/eureka-engineering/understanding-allocations-in-go-stack-heap-memory-9a2631b5035d.

[8] “Static Local Variable in Go.” Stack Overflow, 31 May 2015, stackoverflow.com/questions/30558071/static-local-variable-in-go.

[9] Silva, João Henrique Machado. “Why Go Doesn’t Have Generics - HackerNoon.Com.” Medium, 9 Dec. 2021, medium.com/hackernoon/why-go-doesnt-have-generics-b40ef9e69833.

[10] “Water: How Much Should You Drink Every Day?” Mayo Clinic, 14 Oct. 2020, www.mayoclinic.org/healthy-lifestyle/nutrition-and-healthy-eating/in-depth/water/art-20044256#:%7E:text=About%2015.5%20cups%20(3.7%20liters,fluids%20a%20day%20for%20women.

[11] “Most Complex ‘Hello World’ Program You Can Justify.” Code Golf Stack Exchange, 4 Feb. 2012, codegolf.stackexchange.com/questions/4838/most-complex-hello-world-program-you-can-justify.

[12]“Default Values for Go Types - Ado.Xyz.” Default Values for Go Types, ado.xyz/blog/default-values-for-golang-types/#:%7E:text=The%20default%20or%20zero%20value,string%20(i.e%20%22%22). Accessed 5 Mar. 2022.

[13]“Higher Order Functions in Golang - Golangprograms.Com.” Higher Order Functions in Golang, www.golangprograms.com/higher-order-functions-in-golang.html. Accessed 5 Mar. 2022.

[14] “Go by Example: Goroutines.” Go by Example: Goroutines, gobyexample.com/goroutines. Accessed 5 Mar. 2022.