LET'S GO

A PREPRINT

Gavin Magee
gtmagee@email.arizona.edu

Christian Capriotti
ccapriotti@email.arizona.edu

April 27, 2020

ABSTRACT

Go is a new programming language taking the developer community by storm. The open-source language developed by Google is both simple and efficient. It's easy to learn due to it being based on the C language, but with some modern improvements such as garbage collection. Huge companies like Dropbox and Uber have begun using the language due to it's support for concurrency with a large focus on scalability. [3] The language has built-in modern assets like built in testing, documentation and linting, making it a wonderful programming language to write large projects in. These are just a few reasons Go has become one of the most sought after languages by employers. [6]

1 Introduction

Go is a rapidly growing language designed by Google that was designed from the ground up to be simple, reliable, and efficient. How's it simple? Go has a minimal keyword set, familiar syntax, concurrency primitives, package management, and type inference. How's it reliable? The language was designed for programming at a very large scale; Google made sure there was a large focus on designing Go to be more reliable than other alternatives. How's it efficient? It's extremely fast in comparison to other garbage collected languages like Python. There's a reason Go is one of the top 10 fastest growing languages on Github. [9]

This paper discusses Go's rich history, simple and easy to understand control structures and data types, and really neat concurrency/package management tools.

2 History

Go was designed by a group of Google Engineers: Robert Griesemer, Rob Pike, Ken Thompson, and Russ Cox. The language was created due to frustration with existing languages, as they believed that no single language possessed qualities such as efficient compilation, execution, or general ease of use. The designers saw programmers choosing languages that prioritize ease of use over safety and efficiency, and realized a change needed to be made. [5]

Aimed to integrate the strengths of both interpreted, dynamically typed languages and statically typed, compiled languages; Go is designed to be easy to use, efficient, safe, and modern. This was all achieved by doing things such as formulating intuitive syntax, designing a new type system, incorporating an efficient garbage collector, supporting networked and multi-core computing, and more. [5]

Though the language was conceptualized in late 2007, The Go Programming Language Project was officially launched on November 10, 2009 as an open source project. Go is very much alive, popular applications such as Dropbox, Docker, and Kubernetes have been written in Go, and the language continues to grow in popularity. [5]

3 Control Structures

Go's offerings of control structures is fairly reminiscent of C's; apart from the strange fact that there is no **while** keyword.

3.1 If

To no surprise, the if statement works as any programmer would hope:

```
if x < 0 {
    return -x
}
```

The use of **else if** and **else** are used in Go as well:

```
if x < 0 {
    return -x
} else if x >= 0 {
    return x
} else {
    fmt.Print("Impossible!")
}
```

Go doesn't totally copy C's syntactic rules for if statements, however. Braces are mandatory when writing if statements, even if they're just a single line. Also, Go allows you to add an initialization statement to your if statements: [4]

```
if x, y := 3, 5; x % 2 == 0 || y % 2 == 0 {
    fmt.Print("x or y is even")
}
```

3.2 For

If you're wanting to loop in Go, **for** is your only option. Go's for loop is very similar to C, though not identical. Also, as previously mentioned, Go doesn't have a **while** loop - and that's because **for** handles that for us: [4]

```
// Like a C for
for init; condition; post { }

// Like a C while
for condition { }

// Like a C for(;;) or while(1)
for { }
```

A controversial decision, to say the least.

Here's what a basic for loop in Go would look like:

```
for i := 0; i < n; i++ {
    fmt.Print(i)
}
```

If you want to iterate over an array, string, or map, the use of the **range** clause becomes necessary: [4]

```
array := [3]string{"One", "Two", "Three"}

for index, element := range array {
    fmt.Println(index, "=>", element)
}
```

3.3 Switch

Continuing the pattern, Go's **switch** can be used similarly to C's although it's been designed to be more flexible. Expressions are optional and **switch** works with a multitude of expressions, not just integers/constants. More importantly, use of break isn't necessary to prevent fall through, and multiple cases can be declared in a single line using commas: [4]

```
day := 5
switch day {
case 1:
    fmt.Println("I don't care if Monday's blue")
case 2, 3:
    fmt.Println("Tuesday's gray and Wednesday too")
case 4:
    fmt.Println("Thursday I don't care about you")
case 5:
    fmt.Println("It's Friday I'm in love")
default:
    fmt.Println("Not a weekday")
}
```

The above example prints "It's Friday I'm in love".

3.4 Defer

The **defer** keyword is used to introduce a function call that must be executed when the function returns. For example, the following will print "1,2" due to defer making the function call to Println(2) the last thing executed. [4]

```
defer fmt.Println(2)
fmt.Print("1,")
```

3.5 Panic and Recover

Go also offers a way of altering control flow through functions called **panic()** and **recover()**.

Panic is useful when an error occurs (perhaps due to some unexpected argument) and you wish to let the callers of the function that panicked know about this error. Whenever a function calls panic, the function that called it stops execution, executes any deferred functions, and returns a panicked status to the caller of the function that called panic. This continues up the chain of function calls until the program crashes, unless stopped by another function called recover. Below is an example of panic, where only "Main: 1" and the error message "Panicked!" is printed, along with an error. [4]

```
func main(){
    fmt.Println("Main: 1")
    panic("Panicked!")
    fmt.Println("Main: 2")
}
```

When recover is called, it is able to handle the panic and allow normal execution of the program to continue. An important thing to note is that recover can only be called inside of a deferred function. This is because recover simply returns **nil** if there is no panic occurring, and if there is a panic occurring then the only functions that are called are those that are deferred. When recover is called during a panic, it returns whatever argument was passed to the call of panic. Here is an example of recover that will print "I have recovered after I panicked" with no error. [4]

```
func main (){
    defer func(){
        if r := recover(); r != nil{
            fmt.Println("I have recovered after", r)
        }
    }()
    panic("I panicked")
}
```

4 Data Types

Although Go includes many common data types such as **ints** and **arrays**, the way in which the designers have chosen to implement them is sometimes different from other common languages.

4.1 int

Regarding operators, the **int** works the same as in other languages. However, one noticeable difference between Go and other languages is that Go allows the programmer to clearly decide how many bits to dedicate to represent an textbfint **(8, 16, 32, 64)** and whether or not to make that int signed or unsigned. Examples of this are as shown:

```
var sevenOnes int8 = 0xff >> 1
var sixtyFourOnes uint64 = 0xffffffffffffffff


var a int = 10
var b int = 5
fmt.Println("a + b =", a + b)
fmt.Println("a % b =", a % b)
```

4.2 float

textbfFloat types work just as they do in C. They allow you to add precision to your numerical values. Just like with integers, Go allows you to specify the number of bits (32 or 64) when declaring your textbffloat:

```
var pi float32 = 3.14159
var num float64 = 2,147,483,648
```

These two variables are strongly typed however; so to perform arithmetic, casting would be necessary.

4.3 bool

Following the pattern, **bool** types work much like they do in other languages. You got two options, true or false, have a blast:

```
var a bool = true
var b bool = false
fmt.Println(a && b)
```

Logical operators and (&&), or (||), and not (!) work as one would expect in Go - although their is no explicit XOR operator.

4.4 string

A **string** in Go work as in other languages with the expected operations such as **concatenation, indexing, and slicing**. One interesting thing about strings is that they can be represented by byte arrays, and converted from a byte array back into a string:

```
s := "Funky Town"
```

is equivalent to:

```
s := string([]byte{70, 117, 110, 107, 121, 32, 84, 111, 119, 110})
```

4.5 array

Similar to other languages, an **array** holds a predetermined, unchangeable number of elements all of the same type. What is different from other languages, however, is that an array is passed by value instead of reference. This means that passing an array to a function or storing it in another variable will create a new array and copy the values over from the previous array; any changes made to this copied array will not appear in the original array. How to create an array is shown below:

```
var myArr [2]string = [2]string{"howdy", "partner"}
```

4.6 slice

If you're familiar with Python, the idea of slicing should be familiar. In Go, a **slice** is a dynamically sized type that grabs a "slice" out of a collection:

```
nums := [5]int{10, 20, 30, 40, 50}

var slice []int = nums[1:3]
```

The variable slice now contains the value 10, 20.

4.7 map

Go's **map** is similar to Java's HashMap and Python's dictionary. A map is a data structure containing key/value pairs, where a key maps to a value:

```
m := make(map[string]int)

m["key"] = 1
val := m["key"]
```

The above declares a map m which will store string keys that map to int values. It creates a key "key", and maps it to the value 1. val is then assigned the value mapped to the key "key", which is 1.

4.8 struct

Taken straight out of C, a **struct** is simply a collection of fields in Go:

```
    type Gator struct {
    teeth int
    age int
    name string
}

var myGator Gator = Gator{100, 10, "Jeremy"}
fmt.Println(myGator.teeth)
```

The above creates a new **struct** "myGator", which has 100 teeth, is 10 years old, and is appropriately named "Jeremy". **struct** fields are accessed using the dot notation.

5 Subprograms

Go is rather lacking in the "sub-programs" category. Go mainly relies on functions and packages, and not much more. The combination of packages and structs can be used to mimic what one would tradionally think of as a class. Threads are called Goroutines in Go, and are both extremely easy to use and lightweight.

5.1 functions

Go, like nearly every programming language, supports the use of functions:

```
func main() {}
    fmt.Println(multiply(2, 5))
}

func multiply(a int, b int) int {
    return a*b
}
```

Here the main function calls the multiply function and passes 2 integer arguments, then main prints out the return value of multiply.

Something really cool is that Go supports multiple return values:

```
func main() {}
    a := 1
    b := 2
    a, b = swap(a, b)
}

func swap(a int, b int) int {
    return b, a
}
```

Go also has variadic functions; which take a variable amount of arguments, just like C:

```
    func main() {
        foo(1, 2, 3, 4, 5)
        foo(15)
}

func foo(params ...int) {
    sum := 0

    for i := range params {
            sum += params[i]
    }

    fmt.Printf("sum: %d\n", sum)
}
```

5.2 packages

Every Go program must live within a package. Every small program we've written thus far has had a package declaration at the top of the file:

```
package main

...
```

When we use the **go run** command, the compiler will look for the main function within the main package.

Where packages differ in Go than a language like Java, is there's more utility associated with packages rather than them just being more of an organizational tool. Go doesn't have classes like Java does; so the closest thing to a class in Go would be creating a package with accessible members:

```
package movie

type movie struct {
    name    string
    rating  string
}

func New(name string, rating string) movie {
    m := movie{name, rating}
    return m
}
```

Kind of looks like a Java class, right? It can kind of substitute for one, though it may not be as intuitive.

One could work with this package by creating movie types, and then using associated fields/methods on said "object".

For example:

```
package main
```

```
import "movie"

func main() {
    m := movie.New("The Avengers", "PG-13")
}
```

Above we import the **movie** package, and then call it's **New** method to create a new **movie** variable. Notice how the first letter of the function we called is capitalized - this is important. In Go, symbols with lowercase letters are seen as "private members" to their package, whereas symbols with uppercase letters are seen as "public members".

5.3 goroutines

Goroutine's are how you do simple threading in Go:

```
go foo()
```

starts a new goroutine in the same address space, which will simply run the function:

```
foo()
```

This is extremely intuitive compared to nearly every other programming languages - the ease of concurrency is one of Go's biggest strengths.

6 Summary

Some of Go's biggest strengths are also it's biggest weakness. The simplicity of the language can make programming feel limiting at times, due to there usually only being one or two ways to actually solve a problem in Go. The lack of classes, inheritance, and generics makes the language less complex and lightweight, but can lead to frustration when attempting to design an OOP like program.

Where Go truly outshines the rest is in it's support for concurrency and it's standard library. Writing back-end servers is a natural task in Golang, and is far simpler than what most other languages have to offer. For this reason alone, Go is a wonderful tool to have in your programming toolbox and all programmers should give it a shot to experience the primitive concurrency. Go is continuing to grow at a rapid pace, it won't be long before it's the next "big language".

References

[1] Concurrency. https://www.golang-book.com/books/intro/10.

[2] Data Types. https://golang.org/pkg/go/types/.

[3] Go Users. https://github.com/golang/go/wiki/GoUsers.

[4] Golang Control Structures. https://golang.org/doc/effective_go.html#control-structures.

[5] Golang Origins. https://golang.org/doc/faq#Origins.

[6] Hottest Coding Languages. https://hired.com/state-of-software-engineers#languages.

[7] Packages. https://golang.org/doc/code.html#ImportingLocal.

[8] The Go Blog. https://blog.golang.org/.

[9] The State of the Octoverse. https://octoverse.github.com/.