Go

A Preprint

Christian Capriotti
ccapriotti@email.arizona.edu

Gavin Magee
gtmagee@email.arizona.edu

April 6, 2020

Abstract

1 Introduction

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. [2]

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. [2]

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. [2]

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: [1]

```
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: [1]

```
// 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: [1]

```
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: [1]

```
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. [1]

```
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. [1]

```
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. [1]

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

4 Data Types

5 Subprograms

6 Summary

References

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

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