A INTRODUCTION TO THE GO PROGRAMMING LANGUAGE

A PREPRINT

Jiacheng Yang
Department of Computer Science
The University of Arizona
jiachengyang@email.arizona

Wenkai Zheng
Department of Computer Science
The University of Arizona
wenkaizheng@email.arizona

April 25, 2020

ABSTRACT

Go has been designed to be a language that is easy to use and powerful in current programming. Its control structures are very similar to the control structures of C programming languages. Its if statement and for-loops allows programmers to define local variable, and switches are clearer and more powerful. Its unique keyword defer allows programmers to elegantly handle errors and close resources. Go has the same primitive types, except characters, as other programming languages in C family. It uses rune to represents letters encoded in unicode. It also has structures and interfaces, which provides a way for programmers to define their own data type. It also has the concept of packages, which allows programmers to group code.

***K*eywords** Go · Golang

1 Introduction

As one of the young programming languages, Go has shown its impressive enchantment. In the TIOBE Programming Community index, its highest rank has been tenth, and it is currently positioned twelfth by April, 2020. [3] This shows Go has found its place in the programming world. One possible cause of its popularity can be its C-like syntax as the official website indicates that "Go is mostly in the C family (basic syntax)." [1] Nonetheless, there are some differences and improvements, such as the reduce of the amount of keywords and the special syntax of declarations, made by the designers in order to make code more readable. This paper introduces Go's syntax and compares it to other languages to illustrate the differences.

2 History

Go was designed by Rob Pike, Ken Thompson, and Robert Griesemer in order to achieve fast execution speed and high efficiency of development. Their motivation for designing Go was that they were displeased with programming languages that were over complicated and were not able to easily and safely make concurrent programs.[1] According to the release history in the official website of Go, its first stable version is called r56, which is released in March, 2011.[2] Since 2011, Go has drawn a lot of programmers' attention because of its design philosophy and outstanding ability in concurrent programming. In "Go Developer Survey 2019 Results", Kulesza mentioned that there were about 1 million Go developers worldwide in 2019.[5]

3 Control Structures

Go has five kinds of control structures: if-else, switch, for-loop, labels and defer. If-else, switch, for-loop, and labels work very similar in C. The keyword, defer, is designed for error handling and resource cleaning up.

3.1 Control Structures: if-else

If-else in Go works just like in C with a minor difference that is variables can be defined in the same line of condition statement as the code below. This feature is useful when the condition statement and the body of if-else both need the same value returned by another function. Programmers can just declare a variable to receive it and use it in both places.

```
if x:= somefunction(); x%2 == 0 {
    fmt.Print(x)
    fmt.Println(" is even")
} else {
    fmt.Print(x)
    fmt.Println(" is odd")
}
```

This feature also benefits error handling. For example, when handling errors, since errors in Go are returned by functions, callers can declare variables to receive the return values and then check errors in their condition statements. After that, errors can be handled in the body of if-else. This usage of if-else avoids declaring variables in the name-space outside of the if-else.

```
reader := bufio.NewReader(os.Stdin)
if x, err := reader.ReadString('\n'); err==nil {
    fmt.Println("read: "+ x)
} else {
    fmt.Println("fail")
}
```

3.2 Control Structures: switch

Switch in Go is also slightly different from switch in C. One difference is that programmers no longer need to explicitly use break to break from cases; instead, programmers need to use the keyword "fallthrough" to explicitly execute the case below the current one. For example, if x in the code below is 2 or 3, the program will still print: "x is a prime less than 6."

```
switch x {
case 2:
    fallthrough
case 3:
    fallthrough
case 5:
    fmt.Println("x is a prime less than 6.")
}
```

Another way to accomplish the code above is to provide expected values in a single case as the example below.

```
switch x {
case 2, 3, 5, 7:
    fmt.Println("x is a prime less than 10")
default:
    fmt.Println("x is not a prime less than 10")
}
```

If a variable for comparison is not provided, switch in Go then works just like multiple if-else. Each case is evaluated from top to bottom, and the one evaluated to true will be executed.

```
switch {
case variable%2 == 0:
    fmt.Println("variable is divisible by 2")
case variable%3 == 0:
    fmt.Println("variable is divisible by 3")
case variable%5 == 0:
```

```
    fmt.Println("variable is divisible by 3")
default:
    fmt.Println("variable is not divisible by 2, 3 or 5")
}
```

3.3 Control Structures: for-loop

In Go, iterations are implemented only by for-loops. This means there are no while and do-while loops. However, they can be easily and elegantly simulated by for-loops. A simple for-loop can be implemented as in C.

```
for i := 0; i < 10; i++ {
    sum += i
}
```

When a for-loop is not given initialization and change of variable, it works like a while loop, and programmers do not have to provide extra semicolons as the code below.

```
for i < 10 {
    sum += i
    i++
}
```

For-loop in Go can also work as for-loop in python. Programmers need to use the range keyword in a for-loop as the code below. Range keyword goes over an array or a string and assigns the index to the first variable and the value to the second variable defined in the for-loop. This feature allows programmers to easily iterate over an array or a string.

```
for i, char := range "abcdef"{
    fmt.Printf("Index %d is %s\n",i,string(char))
}
```

3.4 Control Structures: label and goto keyword

Like C, Go supports labels and has goto keyword. Programmers can define a label by a name followed by a colon, and then it can be used by goto keyword. For instance, the code below will skip printing 2 and only print 1 and 3.

```
   fmt.Println("1")
   goto A
   fmt.Println("2")
A:
   fmt.Println("3")
```

Moreover, a loop can be labeled in order to break or continue a specific loop. For example, the code below will only print a = 0, a = 1, and then i = 2.

```
Loop1:
    for a:=0;a < 10; a++ {
        fmt.Println("a = "+ strconv.Itoa(a))
        for i := 1; i < 4; i++ {
            fmt.Println("i = "+ strconv.Itoa(i))
            if i%2 == 0 {
                break Loop1
            }
        }
    }
```

3.5 Control Structures: defer

Go has a unique keyword "defer". As mentioned at the beginning of this section, defer is designed to deal with error handling and resource cleaning up. Programmers can defer a call of function immediately after the return of the current function. For example, the code snippet below will first print zero and then one.

```
func foo() {
    defer func() { fmt.Println("one") }()
    fmt.Println("zero")
}
```

When there are multiple defer's, it follows the rule of FILO (first in last out). So, the code snippet below will print two after one.

```
func deferPrintln() {
    defer func() { fmt.Println("two") }()
    defer func() { fmt.Println("one") }()
}
```

Because a deferred function will always be executed regardless of errors, programmers can close resources, such as IO-stream, in a deferred function as the code below.

```
func writeTo(fileName string) {
    f, _ := os.Create(fileName)
    defer func() {f.Close()}()
    //write file
}
```

4 Data Types

Go is a statically typed language and is almost strongly typed with a package called unsafe that helps programmers to escape from the type system. The built-in types are int, float, rune, byte, string, and pointers. It also has arrays and maps that rescue programmers from building their own.

4.1 Data Types: discrete types

In Go, int, float, rune, and byte are discrete types. Int and float are actually two groups of types. Int consists of int, uint, int8, uint8, int16, uint16, int32, uint32, int64, and uint64. Similarly, float consists of float32 and float64.
The type, rune, is similar to char in other languages; however, rune is designed to store Unicode as Go tends to be a language for international usages. For example, the code below shows a rune stores a Japanese Hiragana letter - は and prints its Unicode number.

```
var p8 rune = は
fmt.Printf("%#U %+q %c\n", p8, p8, p8)
```

Byte represents 8 bits as widely accepted in the computer science world, and it works just as in other language like Python.

4.2 Data Types: composite types

String, array, and map are built-in composite types. Strings are actually an array of rune. However, it is tricky to manually iterated over or indexed strings because strings can contain languages other than English, and they take more indexes than English letters. For example, the string in the code below contains English letters, Chinese letters, and Japanese letters. The index of g in the string is 2 as it is the third letter; however, the index of 日 is 13, and the index of 本 is 16, which means 日 takes 3 indexes to hold it. Thus, it may be a problem to index and slice a string. However, it is fine to use a for-range loop to iterate over a string since for-range was designed to treat strings specially in order to solve the problem.

```
s  = "English中文日本語は"
for i, c := range s {
    fmt.Printf("%d , %c\n", i, c)
}
```

Array is another built-in composite type. Arrays can hold other built-in types, and, as many other languages, arrays can be indexed and sliced. A slice of an array is based-on the original array. Thus, a modification on a slice is also

applied to the original array. A slice can also be constructed by the make function as the code below, and programmers can specify the length and the capacity of the slice.

```
aSlice := make([]string, 5)
aSlice[0] = ""
```

Map is just like diction in python, but the types of keys and values are set in declaration. A map also can be constructed by the make function as the code below.

```
m := make(map[rune]string)
m['0'] = "zero"
m['1'] = "one"
m['2'] = "two"
delete(m, '0')
```

4.3 Data Types: structures and interfaces

Structures and interfaces are types that programmers can define. A structure can be simply defined as the code below.

```
type Student struct {
    Age   int
    Name  string
    Grade int
}
```

Additionally, a structure can be constructed by giving the values as the code below, and, if a value is not given, Go will assign the default value of the type.

```
derrick := Student{22, "Derrick", 100}
frank := Student{Age: 19, Name: "Frank", Grade: 99}
underwood := Student{Age: 21, Name: "Underwood"}
```

Go uses name equivalence for structures, so the code below is illegal and produces a compile time error.

```
type student2 Student
s := Student{Age: 22, Name: "Derrick"}
var s2 student2 = s
```

Structures can also bind with methods. Normally, a function in Go is defined as below.

```
func getName(s Student) string {
    return s.Name
}
getName(s)
```

However, there is another way to define it.

```
func (s Student) getName() string {
    return s.Name
}
s.getName()
```

By defining it in this way, it can be called as a JAVA method. Interface is also a type in Go. Unlike structures, interfaces only contain signatures of methods as the code below.

```
type adder interface {
    add() int
}
type pair struct {
    First  int
    Second int
}
```

```
func (a pair) add() int {
    return a.First + a.Second
}
```

An instance of an interface has to have the corresponding methods. Thus, instances of interfaces are always structures. In the code above, an instance of pair can be an instance of adder.

4.4 Data Types: polymorphism

Go is criticized for being a lack of support for parametric polymorphism. Although interfaces provide a solution to equally treat different structures that have the same methods associated with, this solution cannot work with primitive types since they do not have associated methods. Thus, programmers usually have to write a function multiple times to support for different primitive types.
Inclusion polymorphism on structures can only be done by embedding, which means a structure cannot extend another. Also, interfaces cannot extend and embed each other, but, as long as, two interfaces define exactly the same methods, any instance of them can be converted without any problem.

4.5 Data Types: pointers

All of the types introduced in this section has an associated pointer type that stores the address of the value. However, there is not a void pointer that allows programmers to step around the type system, and the compiler forbids casting a pointer to a different type. Also, there is not any pointer arithmetic operation, and even comparisons between pointers are banned in Go.[6] By these prohibitions, Go avoids being a weakly-typed language like C and is able to implement garbage collection.

5 Subprograms

Go provides several ways to make subprograms. Structures and interfaces are ways to collect and organize data and methods as discussed above. Packages allow programmers to organize their code and source files. Go-routines give programmers the ability to easily write concurrency programs.

5.1 Subprograms: package

The concept of packages is similar to packages in Java. Go forces programmers to use packages. Even if there is a single file, it has to be marked as main. In a package, all variables and functions whose first letter is capitalized are considered public whereas others are private. This feature provides a way to control access from other packages. When dividing programs into packages, it is recommended to organize them as below.

```
-project
    | -src
        | -package1
        |    | -package1Something.go
        | -package2
        |    | -subpackage
        |    |    | -somthing.go
        |    | -package2Something.go
        | -main
        |    | -main.go
```

When a project is organized as above and its GOPATH is set to project folder, programmers can simply run go install to build the program, and the compiler will figure out dependency by itself.

5.2 Subprograms: go-routine

Go-routines are a kind of subprogram in Glang that basically works as threads in other languages. A go program contains at least one go-routine, which is the main function.[4] Another way to initiate a go-routine is to use the "go" keyword with a function call. As said above, go-routines work as threads, so programmers can use go-routines to do concurrency programming. It is also worth to point out that "go-routines are lightweight and programmers can easily create thousands of them"[4].

6 Summary

Go has similar control structures and data types to C. It also has the concept of packages as Java. With limited number of keyword, programmers no longer need to memorize excessive keywords. Goroutines are lightweight and easy to initiate, so Go has been more and more popular in areas that needs concurrent programming.

References

[1] Go: Frequently asked questions (faq).

[2] Go: Pre-go 1 release history.

[3] Tiobe index for april 2020.

[4] Caleb Doxsey. *An Introduction to Programming in Go*. Addison-Wesley, O'Reilly Media, 2016.

[5] Todd Kulesza. Go developer survey 2019 results, Apr 2020.

[6] Rajeev Singh. Playing with pointers in golang, Oct 2019.