Lets Go-lang!

Allison Bolen

Woodring W2018 343

History:

Robert Griesemer, Rob Pike, and Ken Thompson began designing the language in the fall of 2007. Its intentions were to combine the straightforwardness of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language(1a). It was also built to be fast and modern in that it supports networked and multicore computing. The first release was in 2009. Golang has been maintained as an open source project since. It is based off of C with significant influence from Pascal, Modula, Smalltalk, and Python.

Programing Paradigm:

Golang is imperative. It executes commands that change the state of the program.

Golang is object oriented. Sort of. It supports methods and types and basic object oriented programming style, but it doesn't support the typical class hierarchy. It does not implement inheritance, so it is not polymorphic in a traditional sense. Golang is more diet object oriented. It has encapsulation, method types, and interfaces, but its objects are called structs and look like C structs (5). Here is an example of structs in Golang.

Golang is a concurrent language. It has multi-processing, multi-threading and asynchrony. Goroutines are Golangs threading equivalent statements. Think of them as lightweight threads, they start with the allocation and use of a small amount of stack space compared to traditional threads in other languages. A goroutine functions by creating calls that run concurrently with others. Here is an example of a program that prints the numbers one through ten using Goroutines. As you can see there is no special import or library needed to use goroutines, concurrency is part of the Golang runtime library by default. This is because Go was built with concurrency capabilities in mind. Along with goroutines, Golang also has channels which are the connectors or communication links between goroutines(5). It is possible to send values into one goroutine using a channel from another goroutine, although sending and receiving are blocking calls.

Domains of Use:

Go was created at Google and is used internally as well as implemented at companies around the world. From project management platforms like Teamwork, to social media sites such as Twitter, to program packaging systems like Docker, Go is functional in many areas. Go was made to be capable of concurrent programing, as well as managing big data which it does quite well. Due to its concurrency capabilities it is also quite efficient to implement servers in Go(1b). In general Go is good in cases where strong object orientation and overt class hierarchies are not needed.

Data Abstractions:

Golang, was written "to provide the efficiency of a statically typed compiled language with the ease of programming of a dynamic language"(6, pg 77). Thus Golang is a static implicit language that can function and feel like a dynamic implicit language. Dynamic languages type check or bind during runtime ie. Python. Static languages type check or bind at compile time ie. C. Implicit languages infer the type based on use and context, while explicit languages require more obvious declaration. The types are checked at compile time, but can be written as implicit or explicit. Here is an example of Go using both implicit typing and explicit typing.

Golang is considered strongly typed even though it allows implicit declaration. Strongly typed is essentially explicitly stating the type of an object or variable when it is being used in a program. Even though Golang allows implicit declaration it is considered strongly typed because trying to change the type of a variable causes a compile error unless it is using a function like strconv or atoi.

Golang does not support coercion. You can convert between two different types explicitly. It requires explicit casting as shown here. It has to occur with a cast of the type being converted to.

Golang supports data types such as: boolean types (true, false), numeric types (8, 16, 32, 64 bit unsigned integers, 8, 16, 32, 64 bit integers and 32, 64 bit floating point, and 64, 128 bit complex), strings, and derived types (pointer types, array types, structure types, union types, function types, slice types, interface types, map types, channel Types) (7). Examples of declaration for a few of each category listed above are here.

Control Abstractions

In Golang expressions are a computation of a value by applying operators and functions to operands. Where operands are literals, qualified identifiers, or non-blank identifiers such as constants, variables, functions, or expressions grouped by parenthesis.

Literals consist of basic literals, composite literals and function literals. Where basic literals look like int (34, 045, 0xGOOLANG, ...), float (0.30, 0., 1.e+0, ...), imaginary (0i, 012i, 0.1i, ...), rune, and string(" ", "fsfas", "fsad asf ", ...) literals(7). Composite literals look more like array, struct, slice, element, and map types(7). Function literals consist of anonymous functions which can be assigned to variables or directly invoked. An anonymous function looks like this.

Qualified identifiers are identifiers that are associated with a package like cmplx.Sqrt. The identifier is prepended with the package it comes from and neither the package or the identifier can be blank. Constant identifiers, variable identifiers and function identifiers are shown here.

Valid operators in Golang consist of arithmetic (+,-,*,/,++,--,%), relational (==,!=,>,<,<=,>=), logical (&&,||,!), bitwise (&,|,^), assignment (=,+=,-=,<<=,>>=, etc.), and misc. (pointers: *, &) operators(6). The precedence of these operators is: postfix, unary, multiplicative, additive, shift, relational, equality, bitwise AND, bitwise XOR, bitwise OR, logical AND, logical OR, conditional, assignment, comma(6).

Golang has a few different kinds of selection constructs. It has if statements which are composed of boolean expressions followed by executable statements. There are also if else statements which are essentially the same as regular if statements, but they work in conjunction with regular if statements and can cover more than two different outcomes. There are nested if and if else statements which are if or if else statements that lie inside of other if or if else statements providing further options to execute code based on the outcome of the boolean expressions. There are also switch statements which are like condensed if statements, they allow a variable to be tested against different values. Golang also has select statements which function like switch statements, but for channel communications(6). Here are some examples.

Golang only has two forms of looping, recursion and for loops. These can be used in conjunction or in different combinations with themselves or each other like nested for loops. Golang also has continue, break, and goto statements. Continue will cause a loop to skip an iteration, break will cause the looping to stop, and goto will cause a transfer of control to a new/different statement. Golang for loops will iterate over a list or up/down to a number. Nested loops function the same way, but within another loop. Here are some examples.

Golang requires explicit function declarations. You must specify the types you are passing across and the exact type you are returning.

Declarations tell the compiler what is required to make the function work, definitions are the code that goes within a function body. Its what tells the function what to do. Go supports returning multiple things from a function. Go also supports call by value and call by reference. Call by value is when a method copies the actual value of an argument into the formal parameter of the function. Call by reference is when a method copies the address of an argument into the formal parameter. Here are a few examples.

Golang has some scope rules. There are local variables, global variables, and formal parameters. Local variables are only visible to lines of code within the function the variable is declared. Global variables are accessible from any function within the class that the global variable is declared in. Global variables are held for the entirety/whole lifetime of the program while local variables are only present for as long as their portion of the stack is also present. Formal parameters “are treated as local variables within that function and they take preference over global variables”(6). By default GOlang is pass by value, it can be pass by reference if you use pointers though. Here are some examples.

Golang source files are stored in directories called packages. This enables code reusability across applications. Golang has a standard library that consists of packages to pull functions and methods from. Go makes use of import statements. You could pull in an entire package or a single function from a package. Some packages in the standard library are: math (provides constants and basic functions), fmt (I/O functions), flag(command-line flag parsing), net(portable interface for network I/O)(8). Packages from the standard library are available in the GOROOT location.

Since it is an open source language, Go also has many open source packages. This is good because as Go has grown and its functionality honed, people have found common uses for it that may not have been part of the standard implementations. Now it is easier to focus on the primary task of a project instead of creating the whole system on its own. For example Go has a database package part of its Standard Library which implements SQL, however you can get open source packages that implement other forms of databases, like redis or mongodb. When building reusable pieces of code a package is built as a shared library(9). However when you develop executable applications you want to use “package main”. Package main tells the compiler that this will be an executable and the main function will be the entry point. Here is an example of a package main.

Go has a simple error framework built in. It does not support the traditional try/catch syntax of Java and other languages. Go does, however; return an error code from functions as part of a multivariable return statement. Here is an example of the error handling in use. You can make your own error handling functions or use what is returned from an imported function. In the provided example we see the Sqrt function from the math package being used, this function has error handling built with it, however;

to customize the error message we can create our own like we did in the squareRoot function.

Golang does not have classes. The Golang equivalent would be structs. You can define methods on structs as well as giving them variables. These methods have special arguments called receivers(10). Receivers tell the method what type of structure it will be callable from. They function like methods within a typical Java class. Here is an example.

Golang does have support for interfaces. They are name collections of method signatures. In order to implement an interface you just need to implement all the methods you defined within it. Here is an example on how to use interfaces.

Golang has data protection. Encapsulation is handled at the package level. Names that start with lowercase letters can only be accessed within that package. So when importing a package all identifiers that start with uppercase letters are exported and accessible upon import. Packages are Golangs smallest unit of encapsulation.

Overall

Is Glang readable? Golang does not support operator overloading so a “+” can only ever have one meaning and that is addition. That goes for “-,\,%, ...” too. Golang has a relatively small set of keywords, 25 to be exact. Some of the keywords are: break, default, func, interface, select, else import, return, and var. This relatively small set of keywords adds to the readability of the program in that one does not have to remember too many constructs to fluidly read what is happening. GOlang has some multiplicity. There are a few different ways to add amounts to a variable “v = v + 1, v += 1, v++, …”. This can up the difficulty of readability because you would need to remember what all of those do. Golang is fairly orthogonal in that its small set of primitive constructs can be used effectively to produce the structure and flow of a program in an effective manner. Due to the explicit and implicit nature of Golangs variable declaration, how a variable is declared can change the readability of the code. Implicit declarations can be hard to read because you do not know what type it is. The structure and syntax of Golang’s code is fairly straightforward. Keywords cannot be used as variables, code groups are denoted by curly braces, and meaning is apparent in syntax. Overall I would say Golang is a moderately readable language.

Is Golang writeable? Taking into account that a lot of features that affect readability affect writability I would say it is moderately writeable as well. The small number of primitive constructs is easy to remember, the combination of packages and imported functions and the explicit declaration of functions the abstraction of Golang is easy to understand as well. At compile time you would see an error if you were to pass an incorrect data type to a method. Thus it is easy to use functions we haven't written ourselves and expect them to function properly. Golang supports process abstraction so methods are capable of working on any data type sent to them. Golang also supports data abstraction in creating structs and interfaces. Golang is also fairly expensive, calculations and statements can be made concisely. Overall I would say that Golang is fairly writeable.

Is Golang reliable? GOlang type checks at compile time, although it allows implicit declaration it sets variables declared implicitly to an inferred default. Golang supports error handling. Golang does allow aliasing, which can make reliability more difficult. Overall I would say Golang is a reliable language.

Code appendix:

Note: Most of these examples will run on this online compiler.

However, I have found code using Goroutines does not execute well there.

The relevant parts of the example are highlighted in yellow.

Calculator Program: This program is on my github.

Goroutine example:

```go
package main
import "fmt"
func prt(num int){
     fmt.Println(num)
}
func main() {
  // this will be a static call to the prt function
     for i := 1; i < 10; i++ {
          prt(i)
     }
  // this is a concurrent call to the prt function using
goroutines
  for i := 1; i < 10; i++ {
          go prt(i)
     }
  fmt.Println("Golang is made for concurrency.")
}
```

Implicit and Explicit Typing:

```go
package main
import "fmt"
func main() {
     var i int = 1 // explicit integer declaration
     integer := 2  // implicit integer declaration

     var str string = "goodbye" //explicit string declaration
     s := "hello" // implicit string declaration

     fmt.Println(i, s, integer, str) // print their contents
     // integer addition to show they are the same type
     fmt.Println("Sum of i and integer = ", i+integer)
}
```

Structs GoLangs Objects

```go
package main
import "fmt"
type shoes struct{
     size int
     stock bool
     color string
     brand string
}
func main() {
     var runningShoe shoes
     // populating the runningShoe
     runningShoe.size = 10
     runningShoe.stock = false
     runningShoe.color = "blue"
     runningShoe.brand = "Nike"
     // printing the running shoe
     fmt.Println("Size:  ", runningShoe.brand)
     fmt.Println("Color: ", runningShoe.color)
     fmt.Println("Stock: ", runningShoe.stock)
     fmt.Println("Size:  ", runningShoe.size)
}
```

Various Variable Types: (all of it is relevant)

```go
package main
import (
     "fmt"
     "math/cmplx"
)
type boat struct{
     docked bool
}
func main() {
     var integer int = 0
     var lng float64 = 90.9
     var str string = "hello"
     var boolean bool = true
     var unsigned uint64 = 1<<64 - 1
     var imaginary complex128= cmplx.Sqrt(-5 + 12i)
     var skipper boat
     skipper.docked = false
     var arry [2]string
```

```
	arry[0] = "Lets"
	arry[1] = "Go!"

	fmt.Println("integer: ", integer)
	fmt.Println("float: " , lng)
	fmt.Println("string: ", str)
	fmt.Println("boolean: ", boolean)
	fmt.Println("unsigned int 64: ", unsigned)
	fmt.Println("complex: ", imaginary)
	fmt.Println("structure boat: ", skipper.docked)
	fmt.Println("array: ", arry)
}
```

Explicit Type Conversion

```
package main
import (
	"fmt"
	"strconv"
)

func main() {
	i, _ := strconv.Atoi("-42") // you have to
	var s string = strconv.Itoa(-42)
	var f float64 = math.Sqrt(float64(3*3 + 3*3))
	var un uint = uint(f)
	fmt.Println(i)
	fmt.Println(s)
	fmt.Println(f)
	fmt.Println(un)
}
```

Anonymous Function

```
package main
import "fmt"
func main() {
	fmt.Println(func(a, b int) int { return a*b}(3, 5))
}
```

Constants, Variable, Function Identifiers (7)

```
const Pi float64 = 3.14159265358979323846
const zero = 0.0         // untyped floating-point constant
const (
     size int64 = 1024
     eof        = -1  // untyped integer constant
)
const a, b, c = 3, 4, "foo"  // a = 3, b = 4, c = "foo"

var i int
var U, V, W float64
var k = 0
var x, y float32 = -1, -2
var (
     i       int
     u, v, s = 2.0, 3.0, "bar"
)

func min(x int, y int) int {
     if x < y {
          return x
     }
     return y
}
```

Selection Constructs:

```
package main
Import "fmt"

func main() {
     var choice int = 3
     var decide bool = true
     // if statements
     // and if else statements
     // and nested if statements
     if(choice == 2){
          fmt.Println("Golang is Awesome!")
     } else {
          fmt.Println("Golang is cool!")
          if(decide){
               fmt.Println("Golang is cooler than ice cold!!")
```

```
                }
        }
        // switch statements
        var i int = 9
        switch i{
                case 9: fmt.Println("GOOD!")
                case 8: fmt.Println("DECENT!")
                case 7: fmt.Println("ALRIGHT!")
                default: fmt.Println("OH WELL!")
        }
}
```

Iteration:

```
package main
import "fmt"
func back(n int){
        if(n==30){
                fmt.Println("this is the last value of N: ", n)
        } else {
                fmt.Println("this is a value of N: ", n)
                back(n-1)
        }
}
func main() {
        //this will print all the values 0 to 5
        for i := 0; i < 5; i++ {
                        fmt.Printf("value of i: %d\n", i)
        }
        // print all values from 5 to 10 but between each one we
        // will print all the values between 10 and 15
        for i := 5; i < 10; i++ {
                        fmt.Printf("value of i: %d\n", i)
                for j := 10; j < 15; j++ {
                                fmt.Printf("value of j: %d\n", j)
                }
        }
        // recursion print all values from 40 to 30 backwards
        back(40)
}
```

Functions:

```
package main
```

```
import "fmt"
func back(n int){
	if(n==30){
		fmt.Println("this is the last value of N: ", n)
	} else {
		fmt.Println("this is a value of N: ", n)
		back(n-1)
	}
}
func addMulti(a,b,c,d int)(int, int){
	return (a+b),(c*d)
}
func subADDDiv(a,b,c,d *int)int{
	return *a - *b + *c / *d
}
func main() {
	var a,b,c,d int = 2, 4, 6, 8

	// recursion function not return value
	back(40)

	// returning multiple values, call by value
	add, multi := addMulti(2,3,4,5)
	fmt.Println(add, multi)

	// call by reference return single value
	result := subADDDiv(&a,&b,&c,&d)
	fmt.Println(result)
}
```

Scope:

```
package main
import "fmt"
// global variable a
var a, c int = 100, 45

func sum(a,b int)int{
	fmt.Println("Sum 1: ", a)
	// try to print d here the program will break because
	// d is out of scope for this function
	return a+b
}

func main() {
    var a, b, d int = 2, 4, 5

    // this will print 2 because s is a formal parameter
    // and takes precedence over the global vars
    fmt.Println("Main 1: ", a)
    // this won't even print 100
    add := sum(a,b)
    fmt.Println("Main 2: ", add)

    // this will have access to the global variable c because
    // there is no formal parameter with the same name
    fmt.Println("Main 3: ", c)
    // this will print the local variable d
    fmt.Println("Main 4: ", d)
}
```

Executable Main:

```
package main
import "fmt"
func notMain(){
     fmt.Println("It even entered the main func first!")
}
func main() {
     fmt.Println("We have an executable program!")
     notMain()
}
```

Error Handling(6):

```
package main

import( "errors"
     "fmt"
     "math" )

func squareRoot(value float64)(float64, error) {
   if(value < 0){
      return 0, errors.New("Math: negative number passed to
Sqrt")
   }
   return math.Sqrt(value), nil
}
func main() {
   result, err:= squareRoot(-1)

   if err != nil {
      fmt.Println(err)
   } else {
      fmt.Println(result)
   }

   result, err = squareRoot(9)

   if err != nil {
      fmt.Println(err)
   } else {
      fmt.Println(result)
   }
}
```

Golang Receivers:

```
package main
import "fmt"
type shoes struct{
      size int
      stock bool
      color string
      brand string
}
// this is a receiver function, it will only work on shoes structs
func (s shoes) popularity() string{
      if(s.brand == "Nike"){
            return "Very popular"
      } else {
            return "not very popular"
      }
}

func main() {
      var runningShoe shoes
      // populating the runningShoe
```

```
	runningShoe.size = 10
	runningShoe.stock = false
	runningShoe.color = "blue"
	runningShoe.brand = "Nike"
	// printing the running shoe
	fmt.Println("Size:  ", runningShoe.brand)
	fmt.Println("Color: ", runningShoe.color)
	fmt.Println("Stock: ", runningShoe.stock)
	fmt.Println("Size:  ", runningShoe.size)

	// calling the receiver function
	fmt.Println("Popularity level: ", runningShoe.popularity())
}
```

Interfaces:

```
package main
import "fmt"
//interface of shoe
type shoes interface{
	price() float64
	condition() string
}
// this will implement the shoe interface
// by making a method with heel as the receiver
type heel struct{
	brand string
	size int
	height int
	stock bool
}
// this will implement the shoe interface
// by making a method with runningShoe as the receiver
type runningShoe struct{
	brand string
	size int
	stock bool
	color string
}
```

```
func (s runningShoe) price() int{
      if(s.brand == "Nike"){
            return 120
      } else {
            return 70
      }
}
func (s runningShoe) condition() string{
      if(s.brand == "Nike"){
            return "GREAT"
      } else {
            return "FAIR"
      }
}
func (s heel) condition() string{
      if(s.brand == "Gucci"){
            return "GREAT"
      } else {
            return "FAIR"
      }
}
func (s heel) price() int{
      if(s.brand == "Gucci"){
            return 500
      } else {
            return 120
      }
}

func main() {
      var rShoe runningShoe
      var hShoe heel
      // populating the runningShoe
      rShoe.size = 10
      rShoe.stock = false
      rShoe.color = "blue"
      rShoe.brand = "Nike"
      // printing the running shoe
```

```go
	fmt.Println("Size:  ", rShoe.brand)
	fmt.Println("Color: ", rShoe.color)
	fmt.Println("Stock: ", rShoe.stock)
	fmt.Println("Size:  ", rShoe.size)

	// calling the receiver function
	fmt.Println("Price: ", rShoe.price())
	fmt.Println("Condition: ", rShoe.condition())

	hShoe.size = 8
	hShoe.stock = true
	hShoe.height = 6
	hShoe.brand = "Gucci"
	// printing the heel
	fmt.Println("\nSize:  ", hShoe.brand)
	fmt.Println("Height in inches: ", hShoe.height)
	fmt.Println("Stock: ", hShoe.stock)
	fmt.Println("Size:  ", hShoe.size)

	// calling the receiver function
	fmt.Println("Price: ", hShoe.price())
	fmt.Println("Condition: ", hShoe.condition())
}
```

Bibliography:

1. https://golang.org/doc/faq
    a. #history
    b. #Is_Google_using_go_internally
2. https://blog.golang.org/open-source
3. https://kuree.gitbooks.io/the-go-programming-language-report/content/2/text.html
4. https://gobyexample.com/channels
5. http://old.usb-bg.org/Bg/Annual_Informatics/2013/SUB-Informatics-2013-6-076-085.pdf
6. https://www.tutorialspoint.com/go/go_data_types.htm
7. https://golang.org/ref/spec#Expressions
8. https://golang.org/pkg/#stdlib
9. https://thenewstack.io/understanding-golang-packages/
10. https://tour.golang.org/methods/1