GoLang
A Programming Language Case Study

Kyle Jon Aure

Version 1.0, 2018-10-31

Table of Contents

1. Introduction

In the computer science field, programming languages vary almost to the same degree of spoken languages. Like spoken languages, there exist programming languages that are well established and often used, like `Java` and `C++`. Then, there are languages that are rarely spoken and are dying due to unpopularity, like `BLISS` and `Lisp`. In this case study, the language to be examined is `Go`.

The art of language design and the factors that constitute an objectively 'good' language vary. Those who develop programming languages do so for specific purposes, company interests, or to improve an existing language. (Scott, 2016, pp. 7-9) Therefore, this case study will examine the history, support, ease of use, software engineering capabilities, and performance of the `Go` programming language.

2. History

The `Go` programming language was developed by Robert Griesemer, Rob Pike, and Ken Thompson. These programmers started working on `Go` in 2007, and it became a public open source project on November 10, 2009. ("FAQ/Origins," n.a.) Today, it is estimated that there are between 504,000 to 966,000 `Go` programmers, often referred to as `Gophers`. (Cox, 2018)

`Go` was developed in response to the boom of cloud computing in the last 2000's. At the time, developing server software using the traditional programming languages like `C++` and `Java` was becoming increasingly complex. Griesemer, Pike, and Thompson wanted to develop a language that would tackle this problem and make concurrency, garbage collection, and memory management more optimized. Since, the release of `Go` in 2009 the platform has yet to see a leveling off of new users adopting the language. ("Stack Overflow Trends," 2018)

3. Support and Uses

Some of the biggest companies today are using `Go` for their product development, including, Google, Uber, Soundcloud, and Tumblr. (Uygur, 2018) The `Go` programming language has made a big impact on the cloud computing space with the most notable products developed on `Go` being `Docker` and `Kubernetes`. In addition, as a Google product `Go` is being used to develop some of the largest scale software today by Google themselves. (Pike, 2012)

`Go` can be developed on the Windows, MacOS, Linux, and more open source operating systems. (Cōngruì, 2018) Additionally, `Go` can be used to develop software for use on personal computers, servers, and mobile devices. The flexibility of being able to development on multiple operating systems for various platforms is one fo the reasons `Go` has been so successful as a fairly new programming language.

Being a newer language `Go` was developed to be supported by existing text editors and Integrated Development Environments (IDEs). `Vim`, `Visual Studio Code`, and `Atom` are some of the most popular Text Editors with `Go` plugins that support syntax highlighting, debugging, and testing. The more robust IDEs for `Go` have been developed on top of some of the more popular general purpose IDEs. These include `GoLand` built on top of `IntelliJ` and `GoClipse` build on top of `Eclipse`. The massive support for `Go` on platforms that developers already use has made a big impact on the success of this programming language.

4. Ease of Use

One of the primary reasons Go was developed was due to the creators frustration of having to choose between efficient compilation, efficient execution, or ease of programming when choosing a programming language. ("FAQ/Origins," n.a.) Go addressed these issues by attempting to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language. ("FAQ/Origins," n.a.) Go is an imperative programming language that includes themes from the Von Neumann, Object-Oriented, and Scripting language classifications. (Scott, 2016, pp. 11-12)

Go can be compiled using a self-hosted compiler written in Go. The Go Compiler can produce binaries to target all of the major operating systems including mobile operating systems. In addition, Go can be compiled using the GCCgo compiler, which is a front end compiler that uses GCC as the back end. ("FAQ/Origins," n.a.) Lastly, GopherJS is a Go compiler that can target Javascript which allows developers to write front end code that can run on almost all browsers.

4.1. Lexical Elements

Go is written in the same style as the C languages but attempts to improve readability and safety. ("FAQ/Origins," n.a.) Arithmetic operators on numbers are done with the infix notation. Symbols used to perform arithmetic operators are standard for addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). In addition, for integers Go provides bitwise logical operators AND (&), OR (|), XOR (^), and CLEAR (&^). Finally, integers also have arithmetic shift operators for left shift (<<) and right shift (>>).

The arithmetic operators listed above, along with a few more terminals are the starting point for the Go programming language specification. The Go programming language uses the Backus–Naur Form (BNF) for the official language specification. ("FAQ/Origins," n.a.) Below is the Go language specification for identifiers such as those used to name variables, functions, and types.

```
identifier = letter { letter | unicode_digit }
```

The right side of this production starts with a letter meaning that all variables must start with at least 1 letter. Next, the curly brackets {} represent 0 or more of the terminals contained inside. Finally, the vertical bar | represents a logical OR. Therefore, an identifier could be on letter, a letter followed by other letters, a letter followed by numbers, or a letter followed by letters and number.

Comments in Go are the same as both Java and C++ using the double backslash (//) for single line comments and slash-star for multiline comments (/* */). All code preceding the double slash and all code surrounded by the slash-star tokens are removed during compilation. Additionally, the white space characters space, tab, carriage return, and new-lines are also ignored / removed during compilation. The only exception to this is when a new-line character is present at the end of statement production. In these cases, the new-line character will be replaced with a semicolon.

4.2. Types

Go is a statically-typed, compiled language. ("FAQ/Origins," n.a.) The built in types supported by Go include Numbers (byte, int, float), Strings, and Booleans. Go is a statically-typed language, but when declaring a variable the type can be inferred similar to a traditional dynamically-typed language.

```
var x int = 5 //Static representation ①
var y = 6     //Dynamic representation ②
z := 7        //Implied representation ③
```

① Static representation can be used, but it is discouraged since the type is implied by the left hand side.

② Dynamic representation can be used anywhere in a Go program and is the most commonly used.

③ Implied representation can only be used inside of functions or methods for local variables.

4.3. Composite Types

Go supports these eight different composite types; array, struct, pointer, function, interface, slice, map, and channel types. Arrays can be declared by using the square bracket symbol ([]) in front of a variable.

```
var a [5]int                             ①
a[1] = 5                                 ②
b := [5]float32{1.2, 1.3, 1.4, 1.5, 1.6} ③
var c [2][3]int                          ④
```

① Array declaration of 5 integers.

② Setting an element of an array.

③ Declaring and initializing an array of 5 floats.

④ Declaring a 2-D array.

Structures are named elements in Go that have named attributes, but not methods.

```
  type Dog struct {Name string; Breed string;} ①
  d := Dog{Name : "Max", Breed : "Pit Bull"}  ②
  fmt.Println("My dog's name is ", d.Name)     ③
```

① Declaration of the dog structure.

② Initialization of a dog object.

③ Use data from structure.

Functions in Go can be used to declare a single function or a mapping of all functions with the same name, input parameters, and return values.

```
type error func(err, string) nil ①
errFuncs := []error {func(e err, notice string) {fmt.Println(e, notice)}, ②
                     func(e err, notice string) {fmt.Println(notice, e)}, ③
                    }
errFuncs[0](e, "Arithmetic error") ④
```

① error is the name of all functions in the form `func (err, string) nil`

② errFuncs is an instance of error and contains two functions.

③ Each inner function is an independent function that performs different tasks with the same input data.

④ Using a function from a group.

Pointers in `Go` are similar to pointers in `C++` which allows variables to be passed by reference instead of value.

```
func print(i *int) {fmt.Println(*i);} ①

func main() {
  j := 5
  print(&j) ②
}
```

① In a function declaration a pointer is denoted by the asterisk symbol (*).

② When called a variables address is sent to the function using the ampersand symbol (&).

The other composite types are a bit outside of the scope of this paper. Speaking of scope.

4.4. Scope

`Go` uses lexical, or static, scoping. ("FAQ/Origins," n.a.) This is very similar to how scoping works in a language like `Java`. Scope expressly defined by the use the curly brackets `{}`.

```
var a = 1 ①

func main() {
    fmt.Println("Global scope, a is", a)

    var a = 2 ②
    fmt.Println("Local scope, a is", a)
    {
        var a = 3 ③
        fmt.Println("Inner scope, a is", a)
    }
}
```

① Within a function if no local variable is declared yet, global scope is used. a = 1.

② Within a function a local variable will be used instead of global scope. a = 2.

③ The curly brackets `{}` define scope not only for functions, but also at the programmers whim. a =3.

5. Software Engineering

5.1. Testing

`Go` provides natively supported testing using the `testing` package. ("FAQ/Origins," n.a.) In order to run a test, declare a function in the form `func TestXxx (t *testing.T)`. Where the identifier starts with `Test` followed a capital letter and zero or more letters of any capitalization.

```
func add(a int, b int) int { ①
    return a + b
}

func TestAdd(t *testing.T) { ②
    var a = 5
    var b = 5
    var c = add(a, b)
    if c != 11 {
        t.Error("Expected ", 11, " got ", c) ③
    }
}
```

① Function to be tested.

② Test function declaration.

③ Fail condition.

In addition to traditional testing functions, `Benchmark` and `Example` are reserved function keywords that allow programmers to run benchmarks and example code, respectively.

5.2. Modularity

Modularity is important in software engineering since it enables programmers to work on small independent chunks of code. This improves readability, debugging, and issues related to scope. `Go` uses a package and import system similar to `Java` and `C++`. Every file in `Go` must start with a package declaration and optionally imports.

```
package main ①

import ( ②
  "fmt"
  "testing"
)

func main () {
  fmt.Println("Example") ③
}
```

① Package declaration. Main is reserved for the `Go` file that contains the main method. This is the only file that can be used with the command `Go run`.

② Single imports can be declared using the import keyword, and multiple imports can be declared by wrapping them in parentheses.

③ Functions from an imported package can be used with the dot (.) operator.

5.3. Portability

As mentioned in the introduction to this section. `Go` is a compiled language with binary targets per operating system. This makes `Go` less portable then languages that have an intermediary form, such as, java byte-code for `Java`. However, the Go Compiler does allow for cross platform compilation using directives like `+build linux darwin`.

6. Performance

As stated previously, one of the reasons `Go` was developed was for server-side development. In addition, `Go` boats about its ability to utilize multi-core processors more efficiently. As a programming language developed by Google it has already been implemented using large data sets. (Pike, 2012)

`Go` handles concurrency with a technology they call `GoRoutines`. `GoRoutines` allow for a simple approach to write code that utilize multiple threads.

```
func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world") ①
    say("hello")
}
```

① go keyword indicates a function say should be run on a new thread.

The ease of entry into concurrency with Go means that high performing programs can be written with ease right out of the gate. In a performance review of the top server side programming languages Go was found to be the best performing in 2018. (Peabody, 2018) This performance review compared Go, Node.js, PHP, and Java. The test team compared

7. Conclusion

Go is a very powerful programming language. It opens up the doors for rapid development, testing, benchmarking, and compilation. With a focus on runtime performance it is clear why this programming language is being used by some of the biggest technology companies of this generation. However, Go still has some room to improve with many developers criticizing the lack of operator overloading, type interfaces, and immutable data types. Go also has unique features missing from competing languages, such as, garbage collection, lambda expressions, and state of the art exception handling. Overall, Go is a step in the right direction for programming languages and for many projects Go is an exceptional choice.

8. Bibliography

Stack Overflow Trends. (2018). Stack Overflow. Retrieved from https://insights.stackoverflow.com/trends?tags=go%2Chaskell%2Cgroovy

Cox, R. (2018). How Many Go Developers Are There? Research!rsc. Retrieved from https://research.swtch.com/gophercount

Cōngruì, W. (2018). GoLang Minimum Requirements. GoLang. Retrieved from https://github.com/golang/go/wiki/MinimumRequirements

Peabody, B. (2018). Server-side I/O Performance. toptal. Retrieved from https://www.toptal.com/back-end/server-side-io-performance-node-php-java-go

Pike, R. (2012). Go at Google: Language Design in the Service of Software Engineering. Google. Retrieved from https://talks.golang.org/2012/splash.article

Scott, M. L. (2016). *Programming Language Pragmatics*. Morgan Kaufmann Publishing.

Uygur, F. (2018). GoUsers. GoLang. Retrieved from https://github.com/golang/go/wiki/GoUsers

FAQ/Origins. (n.a.). GoLang. Retrieved from https://golang.org/doc/faq#Origins