Kennesaw State University

College of Computing and Software Engineering

DEPARTMENT OF COMPUTER SCIENCE

CS 4308 Concepts of Programming Languages Section W03 Spring Semester

Project Deliverable 2 + 3

Neel Ranawat kranawat@students.kennesaw.edu

4/15/2025

Github Repository for programs:

https://github.com/RanaVat/CS4308-Deliverables-2-3

GO Language Evaluation:

In the realm of popularly used programming languages, Go/Golang is a comparatively simpler language. Go breaks from the mold of other general purpose languages and doesn't attempt to replicate the functionality they have in order to maintain its simplicity and ease of use. While extremely easy and intuitive to pick up, Go's focus on simplicity causes it to lose quite a bit of ground on expressivity, error handling, and functionality. The language lacks concepts and features common in other popular languages such as inheritance, operator and function overloading, and generics.

Go does not provide users with try/catch error handling blocks but rather handles errors with multiple return values.The error handling syntax is verbose and can lead to confusion as it features repetitions of [if err !- nil] code blocks. It is somewhat orthogonal but highly limited to maintain ease of use. The language closely follows the UNIX philosophy and remains to be extremely type-safe, run-time efficient and low cost with one of the simplest syntaxes amongst popular programming languages.

Julia Language Evaluation:

Julia features extremely efficient, simple and easy to learn syntax geared towards scientific/mathematical computing that makes it highly writable and readable. It is written to be familiar to programmers who've worked with MATLAB or R in the past and can natively call . The syntax is overwhelmingly similar to that of most other general purpose programming languages but does feature some unique constructs. It is a moderately orthogonal language that does its best to avoid special cases but does occasionally make exceptions for simplicity or performance's sake.

Julia provides the programmer the ability to declare and use a plethora of primitive, composite, and abstract types that further aids to the language's readability and writability. It also allows programmers to define custom types which greatly enhances expressivity and functionality. The language features a high level of abstraction for methods with Multiple Dispatch which makes maintaining code and using code developed by others so much more intuitive.

Julia uses dynamic type checking combined with Just In Time compilation. This is because the language was built with user-defined types alongside Multiple Dispatch in mind for more efficient computations in mathematical programming. With the nature of the language and its unique constructs, it provides the programmer with extensive error-handling due to being prone to run-time errors. In summary, the focus of "truly generic" functions, custom types, and

Multiple Dispatch gives programmers expressivity combined with performance that is unmatched by most general use programming languages while remaining beginner friendly.

Julia BNF:

<program>= <statement_list>

<statement list>= <statement>

| <statement> <statement_list>

<statement>= <assignment>

| <if_statement>

| <for_loop>

| <while_loop>

| <function_definition>

| <return_statement>

| <expression>

<expression>= <array>

| <boolean>

| <unary_expression>

| <binary_expression>

|<variable>

|<call_function>

|<literal>

<function_defintion>= “function” <function_identifier> “(“ <parameter> “)” <block> “end”

<function_identifier>= <variable>

<parameter>= <variable>

|<variable> “,” <parameter>

<function_call>= <function_identifier> “(” <argument> “)”

<argument>= <expression>

<return_statement>= “return” <expression>

<if_statement>= “if” <expression> <block> “end”

|”if” <expression> <block> “elseif” <expression> <block> “end”

|”if” <expression> <block> “else” <block> “end”

<for_loop>= “for” <variable> “in <range> <block> “end”

<range>= < expression> “:” <expressions>

<while_loop>= “while” <expression> <block> “end”

<block>= <statement_list>

Go BNF:

<program>= <package_declaration> <statement_list>

<package_declaration>= “package” <identifier>

<statement_list>= <statement> | <statement> <statement_list>

<statement>= <declarations>| <simple_statement> | < return_statement> | <if_statement> | <for_statement> | <select_statement> | <expression> | < switch_statement>

<if_statement>= “if” <expression> <block> “else” <block>

<for_statement>= “for” <expression> <for_clause> <range> <block>

<declaration>= <function_declaration> | <variable_declaration> | <type_declaration>

<function_declaration>= “func” <identifier> “(“ <parameter> “)” <block>

<variable_declaration>= “var” <identifier> <type> “=” <expression>

<type_declaration>= “type” <identifier>

<type>= <identifier> <boolean> | <identifier> <numeric> | <identifier> <string> | <identifier> <Array> | <identifier> <Slice> | <identifier> <Interface> | <identifier> <pointer>

<expression>= <operands> | <function_literals> | <index> | <method> | <selectors> | <primary>

<operand>= <literal> | “(“ <expression> “)”

Work Done and Reflection:

I was initially drawn to Go because of its prevalence in networking systems and the cloud. Julia piqued my interest as an interesting mathematical/computing language. I began evaluating Julia’s language with the use of its official documentation. I was unsure if I wanted to choose it as one of my languages for the project deliverable until I stumbled upon its use of Multiple Dispatch. I found the combination of dynamic typing and JIT compilation very impressive. With most languages, there is a general overhead to dynamic typing that hurts performance but Julia does a wonderful job curtailing this trade off. I watched some presentations done by the team that developed Julia and came to appreciate the language’s “truly generic” functions.

Go is a language I’d heard quite a lot about but never actually had the pleasure of learning. It's been touted as a very enjoyable language to work with that's made some programmers fall in love with coding all over again. As I am currently taking a course on parallel and distributed systems, I thought it’d be beneficial to supplement my knowledge with a language built for those exact systems in mind. Go proved to be incredibly simple and one of the most beginner friendly languages I’ve learnt. It’s low latency and concurrency make it quite clear to me why it is such a staple of the web app ecosystem. While Go may not be able to do quite about everything a programmer needs, it is exceptional at what it does do.

My programs for Julia include a program for showcasing the use of a dictionary in Julia and another for showcasing multiple dispatch. My programs for Go include a program for showcasing interprocess communication and another for showcasing the creation of objects in Go. I believe these programs demonstrate a general knowledge of their respective languages and they gave me a foundation to write the partial grammars in BNF alongside the official documentation. If I had the chance to go back in time at the beginning of the project, I would most definitely choose to focus on showcasing concurrency for one of my Go programs. While I completed a Julia program to create, calculate, and plot the multiple derivatives of a function, I didn’t feel like the code was original enough to submit for my deliverable.

Sources:

https://docs.julialang.org/en/v1/

https://go.dev/doc/

https://www.youtube.com/@doggodotjl

https://go.dev/ref/spec#Notation

Source Code:

```
// Name: Neel Ranawat
// KSUID: 000988101
// Class: CS 4308 W03 Spring 2025
// This program demonstrates basic IPC with Go
package main

import (
	"fmt"
	"time"
)

// serves spaghetti
func Server(ch chan<- int) {
	for i := 0; i < 3; i++ {
	ch <- i
	fmt.Printf("Spaghetti cooked and served: %d\n", i)
	time.Sleep(time.Second) // Simulates serving
```

```
        }
        close(ch) // Closes channel
}

func ChinesePhilosopher(ch <-chan int) {
        for num := range ch {
        fmt.Println("Chinese philosopher is philosophizing")
        time.Sleep(500 * time.Millisecond) // Simulates thinking
        fmt.Printf("Spaghetti eaten: %d\n", num)
        }
}

func main() {
        //starts a channel for two
        ch := make(chan int, 2)

        go Server(ch)              //Starts serving in channel
        ChinesePhilosopher(ch) //Starts thinking and eating inchannel

        fmt.Println("The Chinese Philosopher figured it out!")
}
// Name: Neel Ranawat
// KSUID: 000988101
```

```
// Class: CS 4308 W03 Spring 2025

// demonstrates objects in Go

package main


import "fmt"


type Animal struct {

	Name		string

	ScientificName string

	FavoriteArtist string

}


func (p Animal) Description() {

	fmt.Printf("The %s's scientific name is %s and it loves listening to %s\n", p.Name, p.ScientificName, p.FavoriteArtist)

}


func main() {

	person := Animal{Name: "Wolf", ScientificName: "Canis Lupus", FavoriteArtist: "Nina Simone"}

	person.Description()

}

# Name: Neel Ranawat

# KSUID: 000988101
```

```
# Class: CS 4308 W03 Spring 2025
# This program showcases dictionaries in Julia
Storage = Dict(
        "Unit A"=>"Baseball Bats",
        "Unit B"=>"Old Furniture",
        "Unit C"=>"Anime Figurines",
        "Unit D"=>"Computer Parts"
)

# Prints the value of the passed key
Storage["Unit D"]

# Prints all keys and values in the dictionary
for item in Storage
        print(item.first)
        print(" : ")
        println(item.second)
end

# Changes value using key
Storage["Unit D"] = "Modern Art Collection"

# Adds a new pair
```

```
Storage["Unit Z"] = "Classified Government Documents"


# Removes a pair
pop!(Storage, "Unit B")
# Name: Neel Ranawat
# KSUID: 000988101
# Class: CS 4308 W03 Spring 2025
# This program showcases Multiple Dispatch in Julia
# Based on the following youtube video: https://www.youtube.com/watch?v=hesjQz__yb8
# overloaded function give_type
function give_type(x::String)
        return "You gave me a String!"
end

function give_type(x::Int64)
        return "You gave me an Int64!"
end

function give_type(x::Float16)
        return "You gave me a Float16!"
end

function give_type(x::Any)
```

```
        return "You gave me a miscellaneous type!"
end

# Multiple Dispatch

function truly_generic(x)
        println("$x: ", give_type(x))
end

truly_generic(64)
truly_generic("Hello World!")
truly_generic(16.0)
truly_generic(true)
```