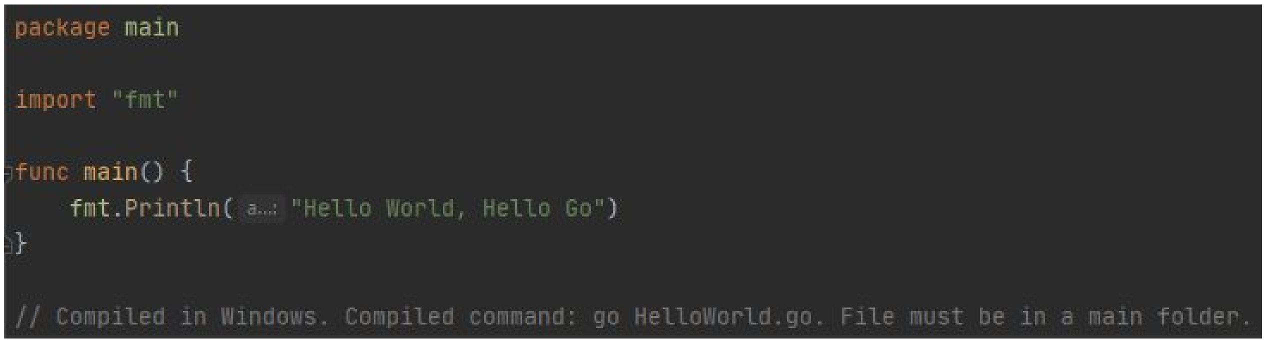
Anh Kieu
CSCI 208 - Final Report
Prof. Wittie

GO Programming Language

*All files are run from main folder. Since Go requires all variables to be used, we will print the variables so that they are used

Introduction

1/ Phase 1: Hello World - How to run Hello World:



```
package main

import "fmt"

func main() {
    fmt.Println( a…: "Hello World, Hello Go")
}

// Compiled in Windows. Compiled command: go HelloWorld.go. File must be in a main folder.
```

2/ Go paradigm: Multi-paradigm, imperative language

(sources: https://kuree.gitbooks.io/the-go-programming-language-report/content/2/text.html)

1/ Background:

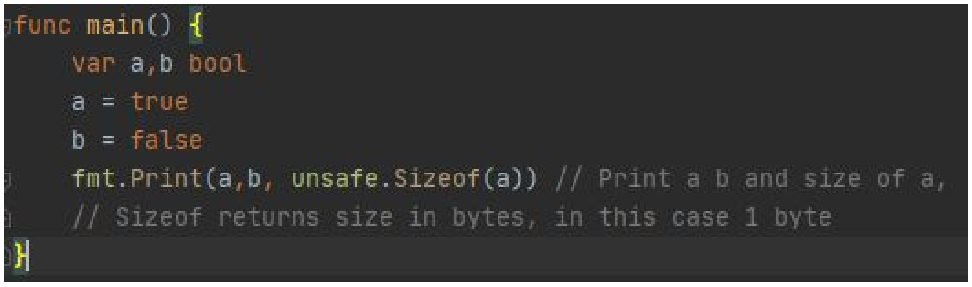
Go is a programming language designed by Google developers, specifically Robert Griesemer, Rob Pike, and Ken Thompson. It was developed in 2007 and was first introduced as an open-source project in 2009. Go is a compiled language. Go was created for Google to start "replacing Python" with a faster language, comparable to C++ while still remains user friendly for Python programmers. Go is also specialized in projects with concurrency for Google's database servers for example [1]. Go is a rising trend these recent years, where more companies are switching from another language to Go [1,2].

Sources:
[1] Introduction - GO for Python Programmers
https://golang-for-python-programmers.readthedocs.io/en/latest/intro.html#what-is-go
[2] Should I Go? The Pros and Cons of Using Go Programming Language
https://hackernoon.com/should-i-go-the-pros-and-cons-of-using-go-programming-language-8c1daf711e46

2/ Primitive Types:

Folder: primitiveTypes

- **Bool:**
  Code (file boolean.go)

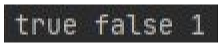


```
func main() {
    var a,b bool
    a = true
    b = false
    fmt.Print(a,b, unsafe.Sizeof(a)) // Print a b and size of a,
    // Sizeof returns size in bytes, in this case 1 byte
}
```

[1]

Compile and run: go run primitiveTypes/boolean.go

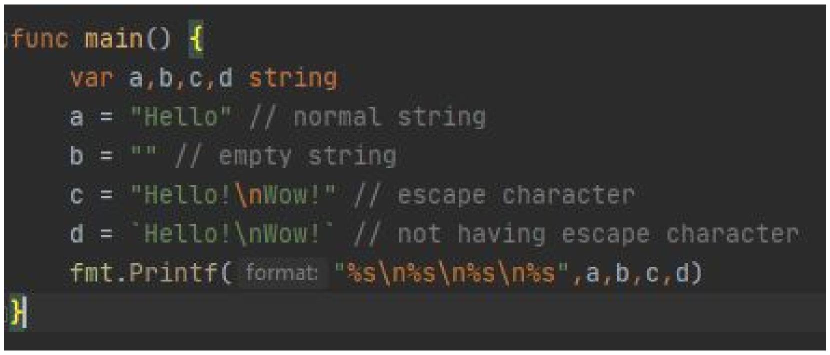
Output:



```
true false 1
```

- **String:**

  Code (file string.go)

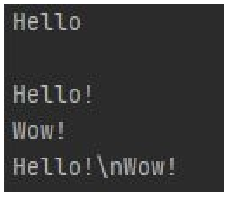


```
func main() {
    var a,b,c,d string
    a = "Hello" // normal string
    b = "" // empty string
    c = "Hello!\nWow!" // escape character
    d = `Hello!\nWow!` // not having escape character
    fmt.Printf( format: "%s\n%s\n%s\n%s",a,b,c,d)
}
```

[1]

Compile and run: go run primitiveTypes/string.go

Output:

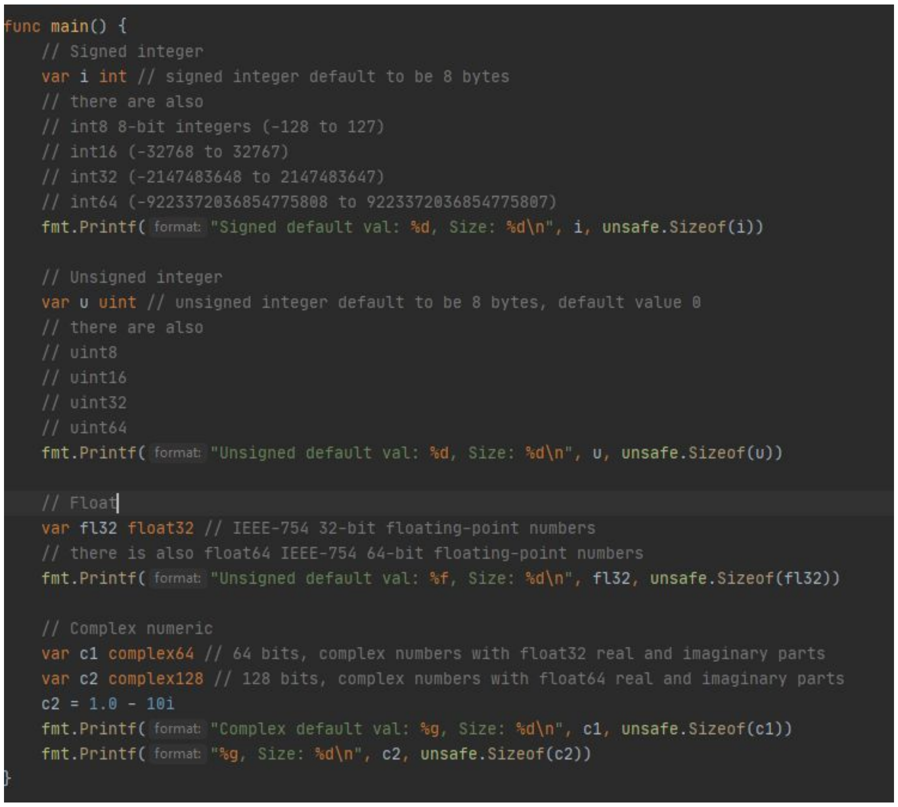


```
Hello

Hello!
Wow!
Hello!\nWow!
```

String is the set of all strings of 8-bit bytes, conventionally but not necessarily representing UTF-8-encoded text. A string may be empty, but not nil. Values of string type are immutable. [1]

- **Numeric type:**

  Code (file numeric.go)

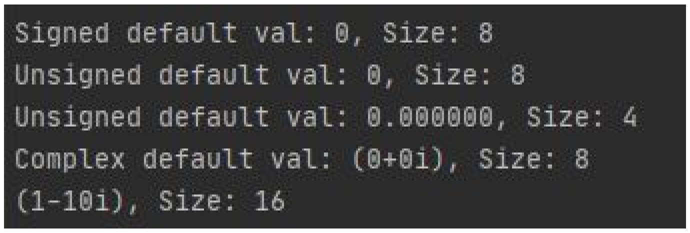


```go
func main() {
    // Signed integer
    var i int // signed integer default to be 8 bytes
    // there are also
    // int8 8-bit integers (-128 to 127)
    // int16 (-32768 to 32767)
    // int32 (-2147483648 to 2147483647)
    // int64 (-9223372036854775808 to 9223372036854775807)
    fmt.Printf( format: "Signed default val: %d, Size: %d\n", i, unsafe.Sizeof(i))

    // Unsigned integer
    var u uint // unsigned integer default to be 8 bytes, default value 0
    // there are also
    // uint8
    // uint16
    // uint32
    // uint64
    fmt.Printf( format: "Unsigned default val: %d, Size: %d\n", u, unsafe.Sizeof(u))

    // Float
    var fl32 float32 // IEEE-754 32-bit floating-point numbers
    // there is also float64 IEEE-754 64-bit floating-point numbers
    fmt.Printf( format: "Unsigned default val: %f, Size: %d\n", fl32, unsafe.Sizeof(fl32))

    // Complex numeric
    var c1 complex64 // 64 bits, complex numbers with float32 real and imaginary parts
    var c2 complex128 // 128 bits, complex numbers with float64 real and imaginary parts
    c2 = 1.0 - 10i
    fmt.Printf( format: "Complex default val: %g, Size: %d\n", c1, unsafe.Sizeof(c1))
    fmt.Printf( format: "%g, Size: %d\n", c2, unsafe.Sizeof(c2))
}
```

[1]

Compile and run: go run primitiveTypes/numeric.go

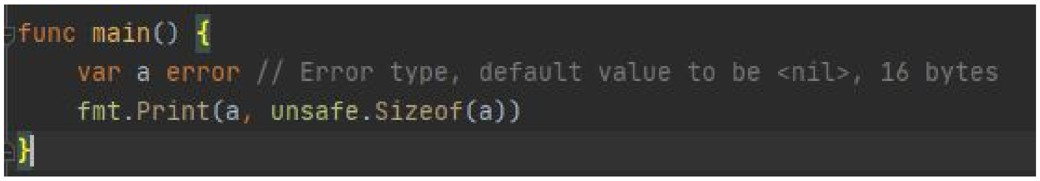
Output:



```
Signed default val: 0, Size: 8
Unsigned default val: 0, Size: 8
Unsigned default val: 0.000000, Size: 4
Complex default val: (0+0i), Size: 8
(1-10i), Size: 16
```

- **Error type:**

  Code (file error.go)

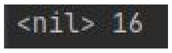


```go
func main() {
    var a error // Error type, default value to be <nil>, 16 bytes
    fmt.Print(a, unsafe.Sizeof(a))
}
```

[1]

Compile and run: go run primitiveTypes/error.go

Output:



```
<nil> 16
```

Sources:

[1] The Go Programming Language Report

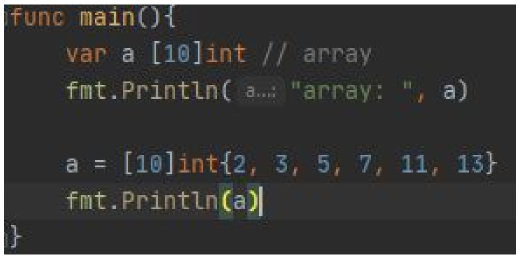
https://kuree.gitbooks.io/the-go-programming-language-report/content/3/text.html

3/ Composite and Constructed Types:

Folder: compTypes

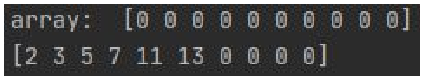
1. Array

An array is a numbered sequence of elements of a single type, called the element type. The number of elements is called the length and is never negative. All the indexes should be non-negative integers and all the entries should be the same type. Array is a mapping in Go with integer as indexes [1]
Code: (file array.go)



```
func main(){
    var a [10]int // array
    fmt.Println( a...: "array: ", a)

    a = [10]int{2, 3, 5, 7, 11, 13}
    fmt.Println(a)
}
```

[1,2]
Compile and run: go run compTypes/array.go
Output:

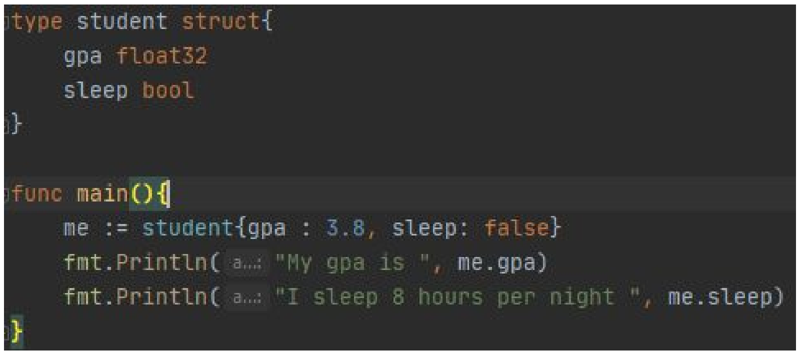


```
array:  [0 0 0 0 0 0 0 0 0 0]
[2 3 5 7 11 13 0 0 0 0]
```

=> Default values in int array are set to be 0
Source:
[1] The Go Programming Language Report https://kuree.gitbooks.io/the-go-programming-language-report/content/4/text.html
[2] A Tour of Go https://tour.golang.org/moretypes/6

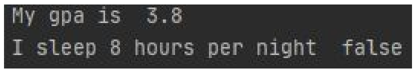
2. Struct

A struct is a sequence of named elements, called fields, each of which has a name and a type. Field names may be specified explicitly (IdentifierList) or implicitly (AnonymousField). Within a struct, non-blank field names must be unique. The struct is the cartesian product type in Go. Each element should have a name and a type. Similar to C, struct doesn't have methods inside the declaration. In other words, methods are defined on structs, not within structs. [1]
Code: (file struct.go)



```
type student struct{
    gpa float32
    sleep bool
}

func main(){
    me := student{gpa : 3.8, sleep: false}
    fmt.Println( a...: "My gpa is ", me.gpa)
    fmt.Println( a...: "I sleep 8 hours per night ", me.sleep)
}
```

Compile and run: go run compTypes/struct.go
Output:



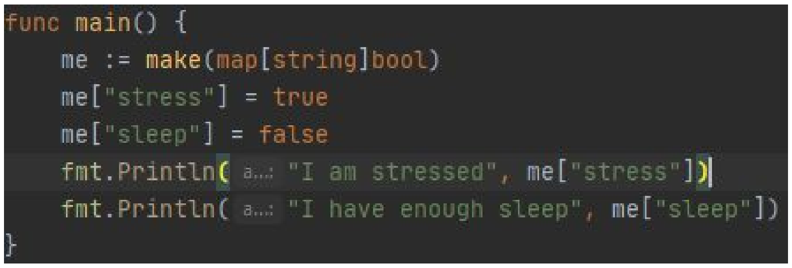
```
My gpa is  3.8
I sleep 8 hours per night  false
```

Sources:
[1] The Go Programming Language Report https://kuree.gitbooks.io/the-go-programming-language-report/content/4/text.html

3. **Map**

A map is an unordered group of elements of one type, called the element type, indexed by a set of unique keys of another type, called the key type. The value of an uninitialized map is nil. Maps are the dictionary in other languages. [1]

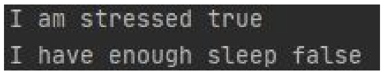
Code: (file map.go)



```
func main() {
    me := make(map[string]bool)
    me["stress"] = true
    me["sleep"] = false
    fmt.Println( a...: "I am stressed", me["stress"])
    fmt.Println( a...: "I have enough sleep", me["sleep"])
}
```

Compile and run: go run compTypes/map.go

Output:



```
I am stressed true
I have enough sleep false
```

Sources:

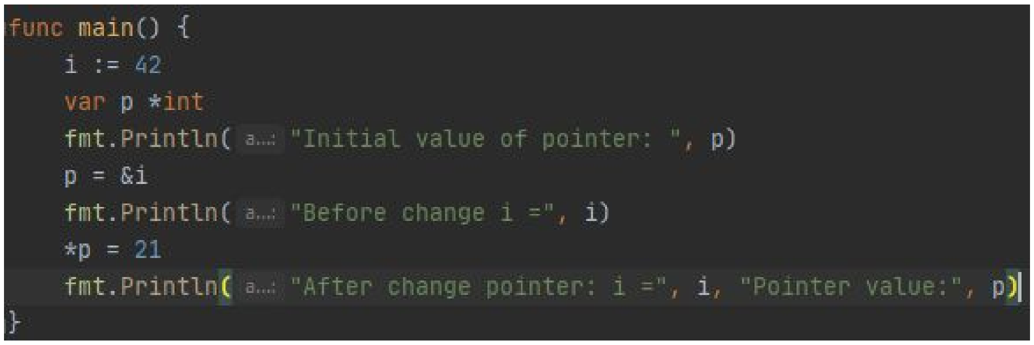
[1] The Go Programming Language Report

https://kuree.gitbooks.io/the-go-programming-language-report/content/4/text.html

4. **Pointer/Reference**

A pointer type denotes the set of all pointers to variables of a given type, called the base type of the pointer. The value of an uninitialized pointer is nil. [1]

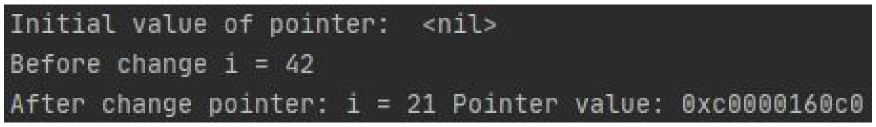
Code: (file pointer.go)



```
func main() {
    i := 42
    var p *int
    fmt.Println( a...: "Initial value of pointer: ", p)
    p = &i
    fmt.Println( a...: "Before change i =", i)
    *p = 21
    fmt.Println( a...: "After change pointer: i =", i, "Pointer value:", p)
}
```

Compile and run: go run compTypes/pointer.go

Output:



```
Initial value of pointer:  <nil>
Before change i = 42
After change pointer: i = 21 Pointer value: 0xc0000160c0
```

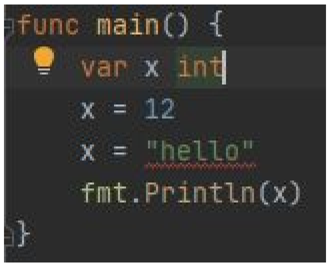
Sources:

[1] The Go Programming Language Report

https://kuree.gitbooks.io/the-go-programming-language-report/content/4/text.html

4/ Static and Dynamic Typing:

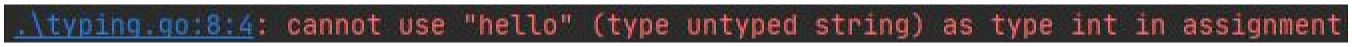
Code (file typing.go)



```
func main() {
    var x int
    x = 12
    x = "hello"
    fmt.Println(x)
}
```

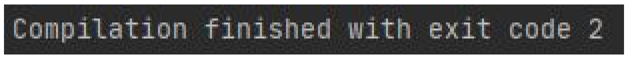
Compile and run: go run typing.go

Output:



```
.\typing.go:8:4: cannot use "hello" (type untyped string) as type int in assignment
```

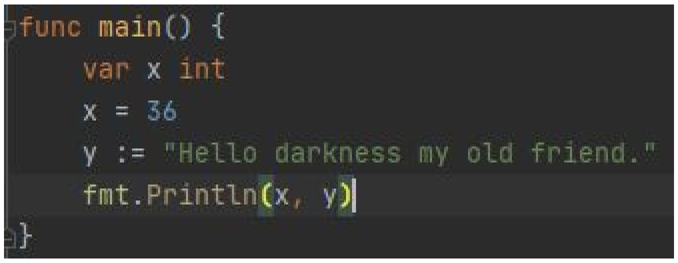
Hence, this language is statically typed. The default values are explained in primitive types question. There is no way to change it to dynamically typed. The check is done in compilation time.



```
Compilation finished with exit code 2
```

5/ Implicitly or explicitly typed:

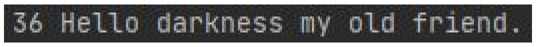
Code (file imexType.go)



```
func main() {
    var x int
    x = 36
    y := "Hello darkness my old friend."
    fmt.Println(x, y)
}
```

Compile and run: go run imexType.go

Output:



```
36 Hello darkness my old friend.
```

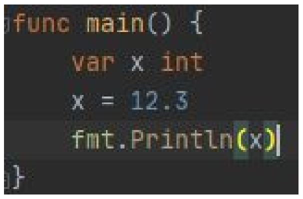
Go could be implicitly or explicitly typed. Its type could be inferred using ":=". There is no difference in any way of declaration. ":=" can only be used in functions. [1]

Sources:

[1] A Tour of Go https://tour.golang.org/basics/10

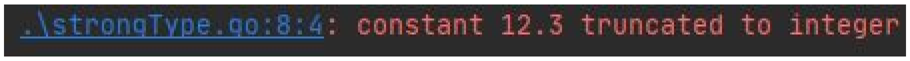
6/ Strongly typed or weakly typed:

Code: strongType.go



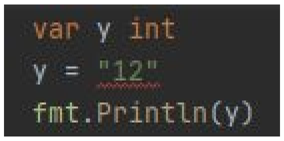
```
func main() {
    var x int
    x = 12.3
    fmt.Println(x)
}
```

Output:



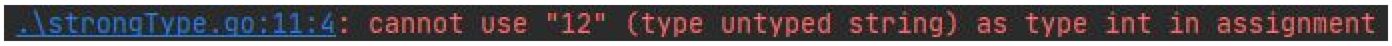
```
.\strongType.go:8:4: constant 12.3 truncated to integer
```

=> Can't do implicit cast from float to int



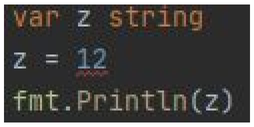
```go
var y int
y = "12"
fmt.Println(y)
```

Output:

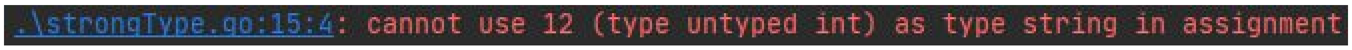


```
.\strongType.go:11:4: cannot use "12" (type untyped string) as type int in assignment
```

=> Can't do implicit cast from string to int

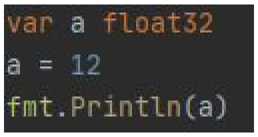


```go
var z string
z = 12
fmt.Println(z)
```

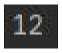


```
.\strongType.go:15:4: cannot use 12 (type untyped int) as type string in assignment
```

=> Can't do implicit cast from int to string

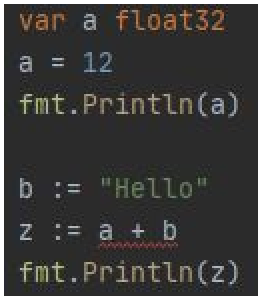


```go
var a float32
a = 12
fmt.Println(a)
```

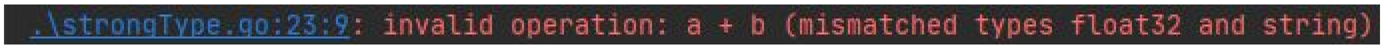
=> 

```
12
```

=> Can do implict cast from int to float



```go
var a float32
a = 12
fmt.Println(a)

b := "Hello"
z := a + b
fmt.Println(z)
```

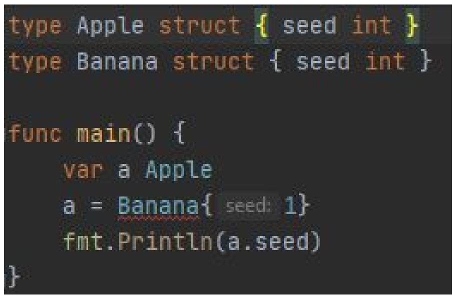


```
.\strongType.go:23:9: invalid operation: a + b (mismatched types float32 and string)
```

=> Can't do concatenate for string and float
Hence, Go is fairly strongly typed. It doesn't allow lossy implicit coercion.

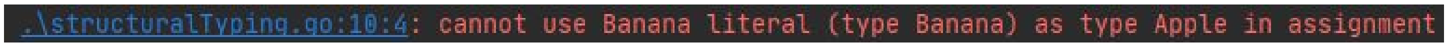
7/ Nominal vs Structural typing:

Code: structuralTyping.go



```go
type Apple struct { seed int }
type Banana struct { seed int }

func main() {
    var a Apple
    a = Banana{ seed: 1}
    fmt.Println(a.seed)
}
```

Compile and run: go run structuralTyping.go
Output:

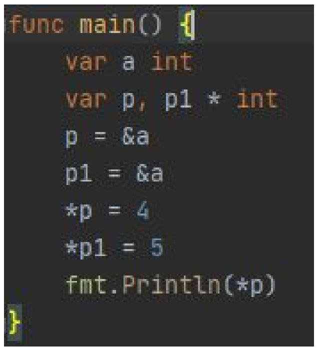


```
.\structuralTyping.go:10:4: cannot use Banana literal (type Banana) as type Apple in assignment
```

=> Structural Typing instead of Nominal

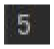
8/ Memory hazard: Alias, dangling pointers, memory leak:

There exists alias hazard: Two pointers can point to the same memory address and change it
Code: file hazard.go



```
func main() {
    var a int
    var p, p1 * int
    p = &a
    p1 = &a
    *p = 4
    *p1 = 5
    fmt.Println(*p)
}
```

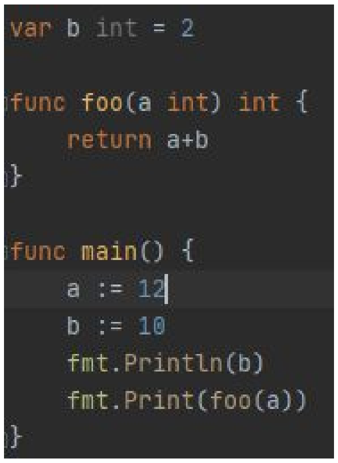
Compile and run: go run hazard.go

Output: 

```
5
```

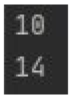
9/ Static vs Dynamic Scope

Code: file scope.go



```
var b int = 2

func foo(a int) int {
    return a+b
}

func main() {
    a := 12
    b := 10
    fmt.Println(b)
    fmt.Print(foo(a))
}
```

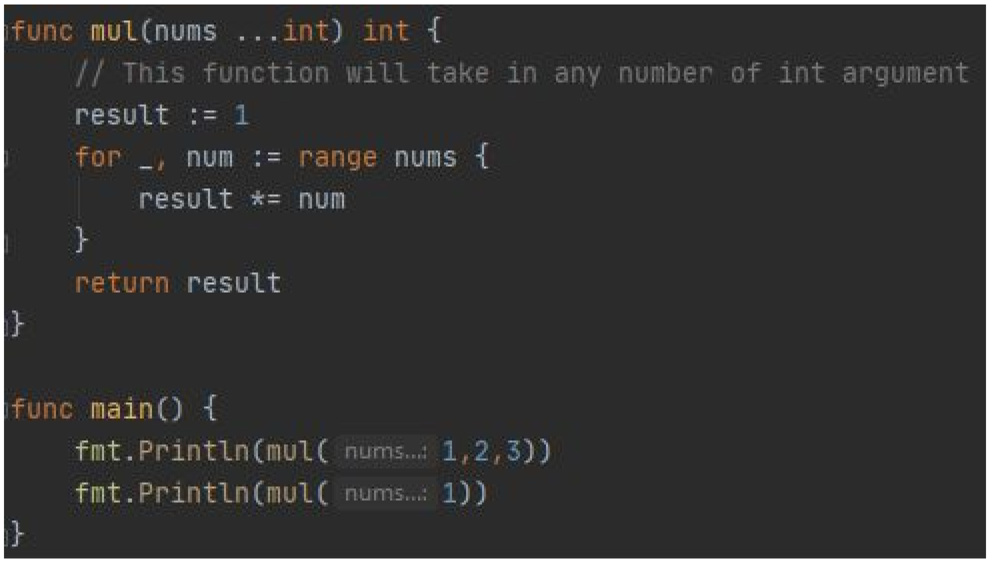
Compile and run: go run scope.go

Output:  => Static scope as b in foo is not b in main but the global one

```
10
14
```

10/ Variadic function (instead of Default parameters)

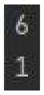
If the **last parameter** of a function has type ...T it can be called with **any number** of trailing arguments of type T. [1]
Code: file variadic.go



```go
func mul(nums ...int) int {
    // This function will take in any number of int argument
    result := 1
    for _, num := range nums {
        result *= num
    }
    return result
}

func main() {
    fmt.Println(mul( nums...: 1,2,3))
    fmt.Println(mul( nums...: 1))
}
```

Compile and run: go run variadic.go

Output:

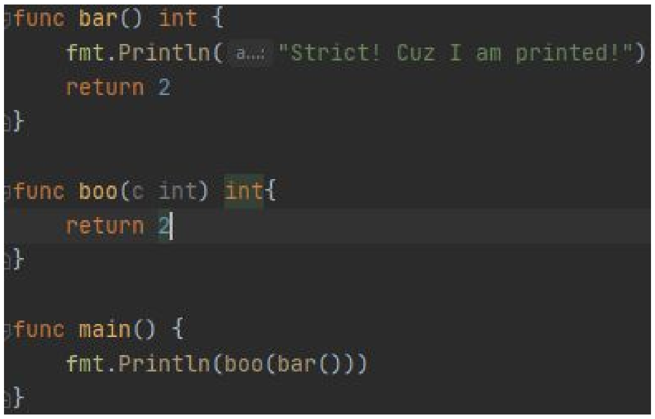


```
6
1
```

Source: [1] Variadic function in Golang https://yourbasic.org/golang/variadic-function/

11/ Strict vs Non-strict evaluation

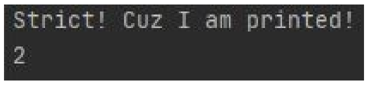
Code: file strict.go



```go
func bar() int {
    fmt.Println( a...: "Strict! Cuz I am printed!")
    return 2
}

func boo(c int) int{
    return 2
}

func main() {
    fmt.Println(boo(bar()))
}
```

Compile and run: go run strict.go

Output:

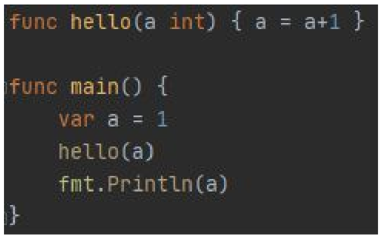


```
Strict! Cuz I am printed!
2
```

Hence, GO uses strict evaluation

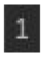
12/ Pass by value or reference?

Code: file pbv.go



```go
func hello(a int) { a = a+1 }

func main() {
    var a = 1
    hello(a)
    fmt.Println(a)
}
```

Compile and run: go run pbv.go

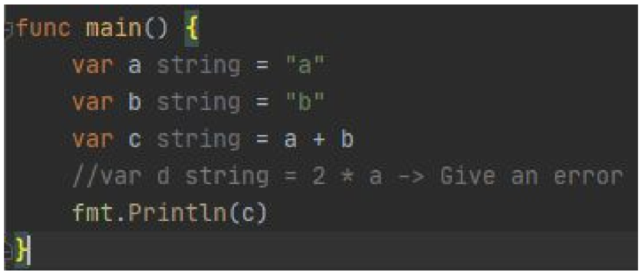
Output: 

```
1
```

=> Sine a does not change, Go passes parameters by value

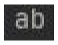
13/ String type

String is a primitive type in Go as mentioned above. String in Go is immutable [1]. Go only offers string concatenation:
Code: file stringType.go - reference from [1]



```go
func main() {
    var a string = "a"
    var b string = "b"
    var c string = a + b
    //var d string = 2 * a -> Give an error
    fmt.Println(c)
}
```

Compile and run: go run stringType.go

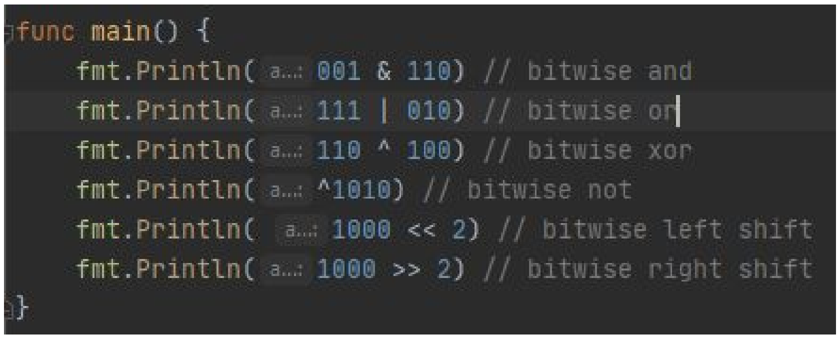
Output: 

```
ab
```

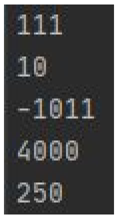
Sources:
[1] The Go Programming Language Report
https://kuree.gitbooks.io/the-go-programming-language-report/content/29/text.html

14/ Math operations (bitwise)

Go offers different bitwise operation:
Code: bitOp.go



```go
func main() {
    fmt.Println( a...: 001 & 110) // bitwise and
    fmt.Println( a...: 111 | 010) // bitwise or
    fmt.Println( a...: 110 ^ 100) // bitwise xor
    fmt.Println( a...: ^1010) // bitwise not
    fmt.Println( a...: 1000 << 2) // bitwise left shift
    fmt.Println( a...: 1000 >> 2) // bitwise right shift
}
```

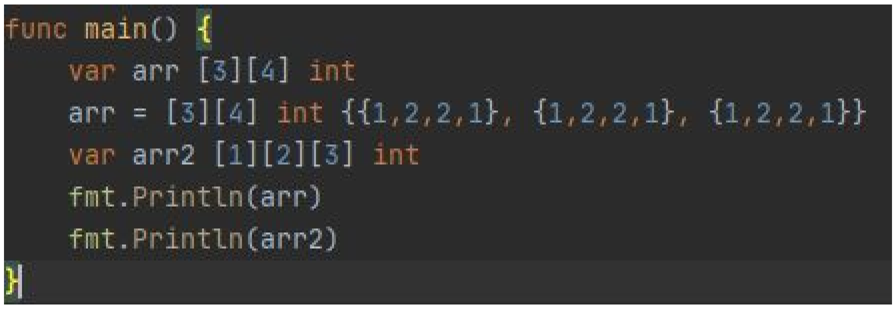
Output: 

```
111
10
-1011
4000
250
```

Compile and run: go run bitOp.go

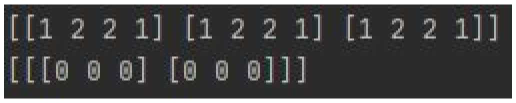
15/ Multi dimension array

Go offers multi-dimension arrays [1]
Code: file mulDem.go



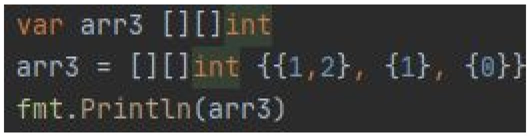
```
func main() {
    var arr [3][4] int
    arr = [3][4] int {{1,2,2,1}, {1,2,2,1}, {1,2,2,1}}
    var arr2 [1][2][3] int
    fmt.Println(arr)
    fmt.Println(arr2)
}
```

Compile and run: go run mulDem.go

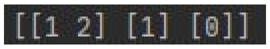
Output: 

```
[[1 2 2 1] [1 2 2 1] [1 2 2 1]]
[[[0 0 0] [0 0 0]]]
```

It also offers jagged multidimension array:
Code:



```
var arr3 [][]int
arr3 = [][]int {{1,2}, {1}, {0}}
fmt.Println(arr3)
```

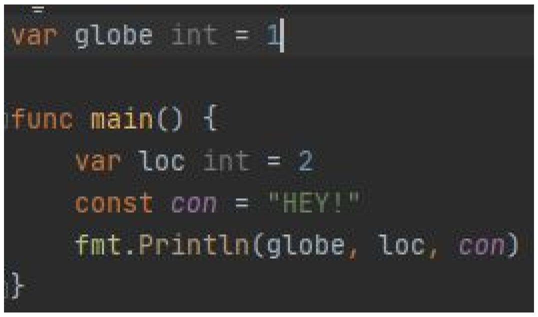
Output: 

```
[[1 2] [1] [0]]
```

Sources:
[1] Tutorialspoint - https://www.tutorialspoint.com/go/go_multi_dimensional_arrays.htm

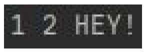
16/ Variables

There are global variable, local variable and constant
Code: file var.go



```
var globe int = 1

func main() {
    var loc int = 2
    const con = "HEY!"
    fmt.Println(globe, loc, con)
}
```

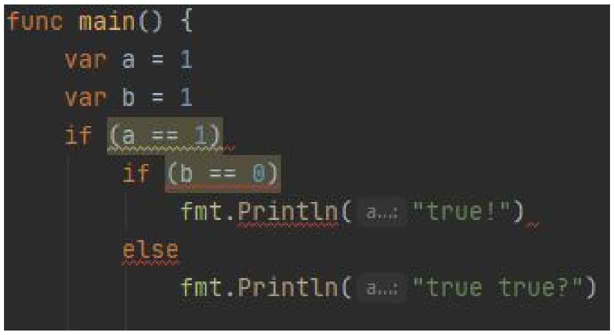
Compile and run: go run var.go

Output: 

```
1 2 HEY!
```

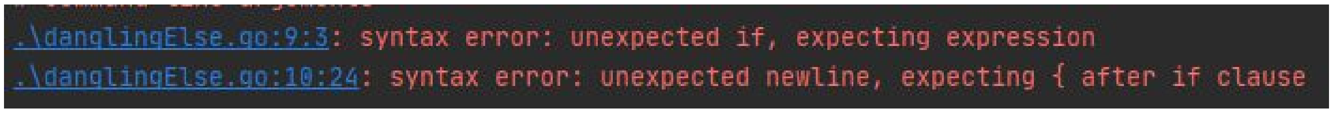
17/ Dangling Else

There cannot be dangling else in Go because all if - else blocks have to be surrounded by bracket { }. The statement after if does not always need parenthesis around. [1]
Code: danglingElse.go



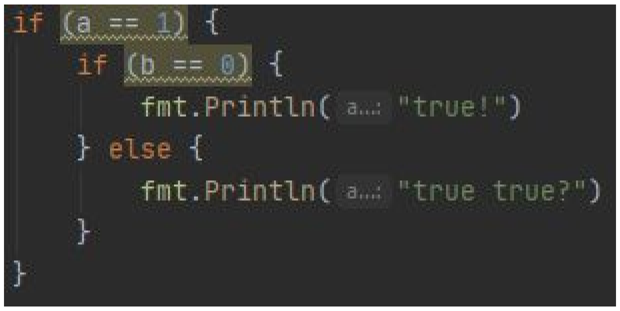
```go
func main() {
    var a = 1
    var b = 1
    if (a == 1)
        if (b == 0)
            fmt.Println( a...: "true!")
        else
            fmt.Println( a...: "true true?")
```

Compile and run: go run danglingElse.go
Output:

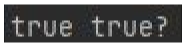


```
.\danglingElse.go:9:3: syntax error: unexpected if, expecting expression
.\danglingElse.go:10:24: syntax error: unexpected newline, expecting { after if clause
```

Correct syntax:



```go
if (a == 1) {
    if (b == 0) {
        fmt.Println( a...: "true!")
    } else {
        fmt.Println( a...: "true true?")
    }
}
```

Output: 

```
true true?
```

Sources:
[1] Go by example - https://gobyexample.com/if-else

18/ Coercion

	int	float	bool	string
int	\	Explicit		
float		\		
bool			\	
string				\

19/ Garbage Collector

20/ Short Circuit Eval

21/ Portability:

Unfortunately, for the performance concern, Go will compile your source code into binary code targeted at your platform. In this sense Go is not portable. [1] However, it does allow cross-platofrm compilation for different architecture and operating systems using the same source code on the same machine. Go tooling provides tools to convert code from one platform to another. [2]

Source:
[1] The Go Programming Language Report
https://kuree.gitbooks.io/the-go-programming-language-report/content/30/text.html
[2] Why Golang Is Great for Portable Apps -
https://codeburst.io/why-golang-is-great-for-portable-apps-94cf1236f481

22/ Error handling:

23/ Imperative feature: Statement vs Expression vs Function vs Procedure

24/ Imperative feature: Pointers/References:

25/ Imperative feature: Stack/Heap

26/

27/

28/ Borrowed idea from Functional: Higher-order functions:

29/

30/

31/ Borrowed idea from OOP: Classes (in Go it is struct)