CS188
Week 1

Spyros Mastorakis

Outline

- Intro
- Go overview
- Assignment 1

Intro

- TA: Spyros Mastorakis
- Email: last_name <at> cs.ucla.edu
- Office Hours: Tuesday 9 – 11AM (366 Engineering VI)

Go Programming Language

- The idea of building Go arose while Google employees were waiting for a Google server to compile

- Compile time was sooooo long…

- So they decided to create a language that works well for building large scale codebases

Syntax

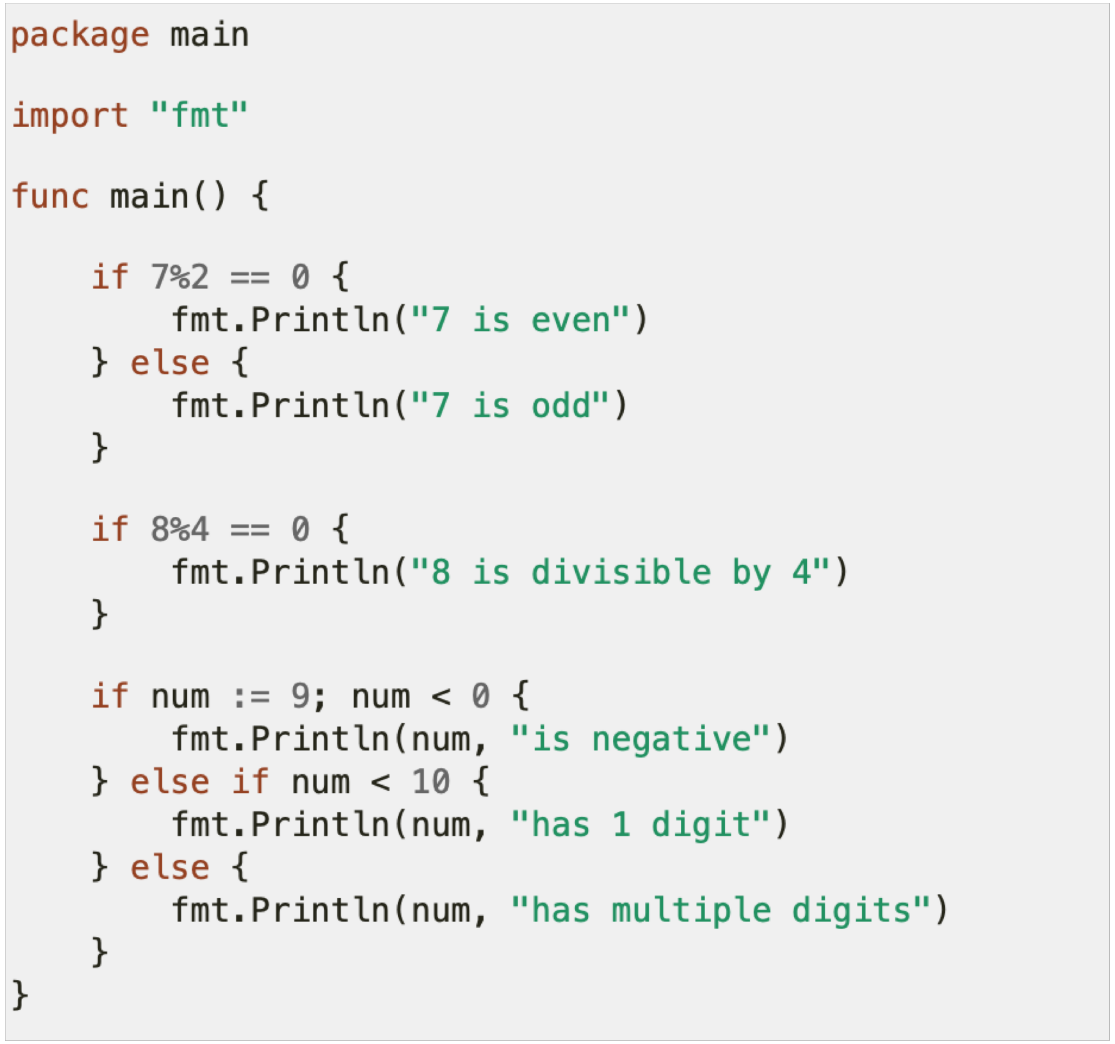
C Syntax	Go Syntax
int x;	x int
int* p;	p *int
int a[3];	a [3]int
int main(int argc, char *argv[])	func main(int, []string) int

Packages (aka header files in C++)

- import "fmt”
  - Contains functions related to formatting and output to the screen

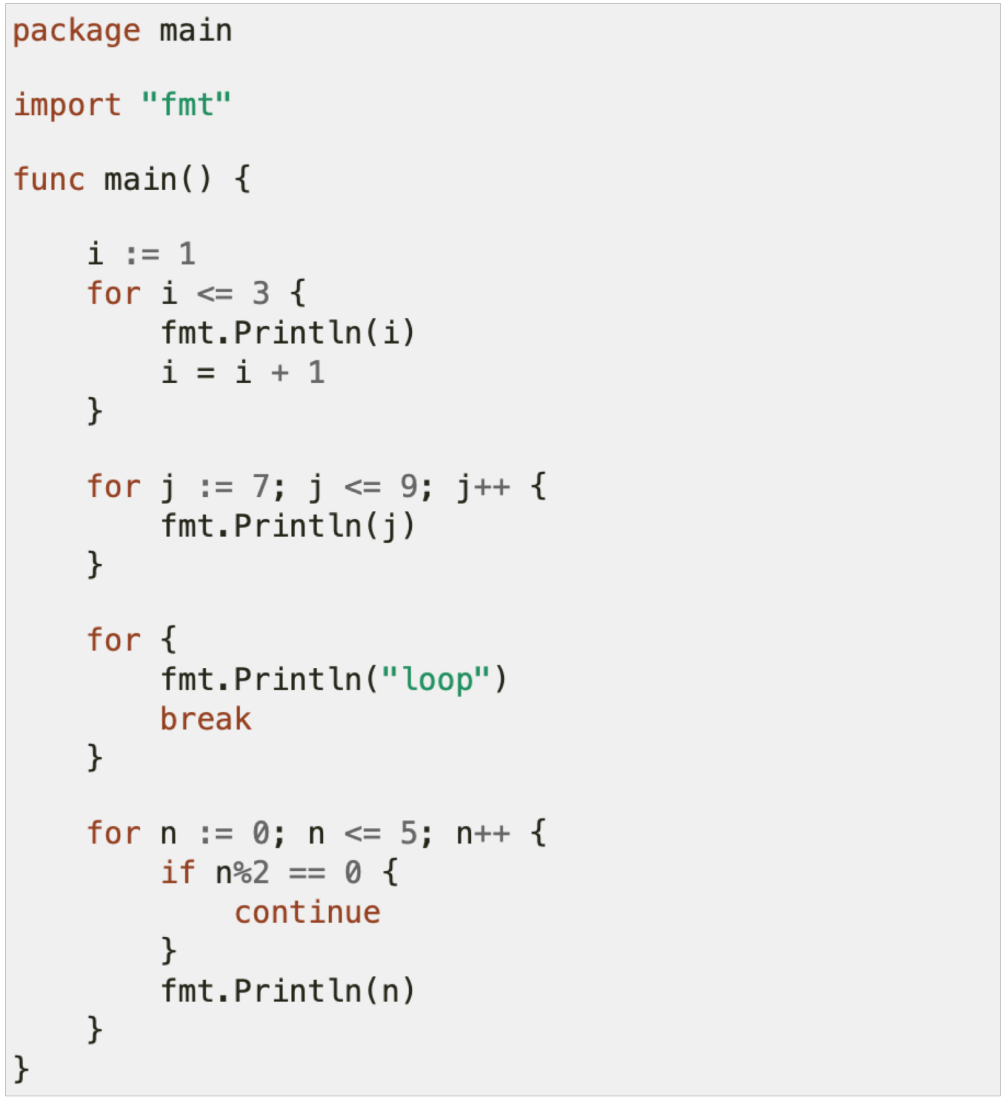
- Why using packages?
  - Better code organization to easily find code that we want to re-use
  - Optimizes the compiler execution by requiring re-compilation of smaller chunks of the program

If Statements



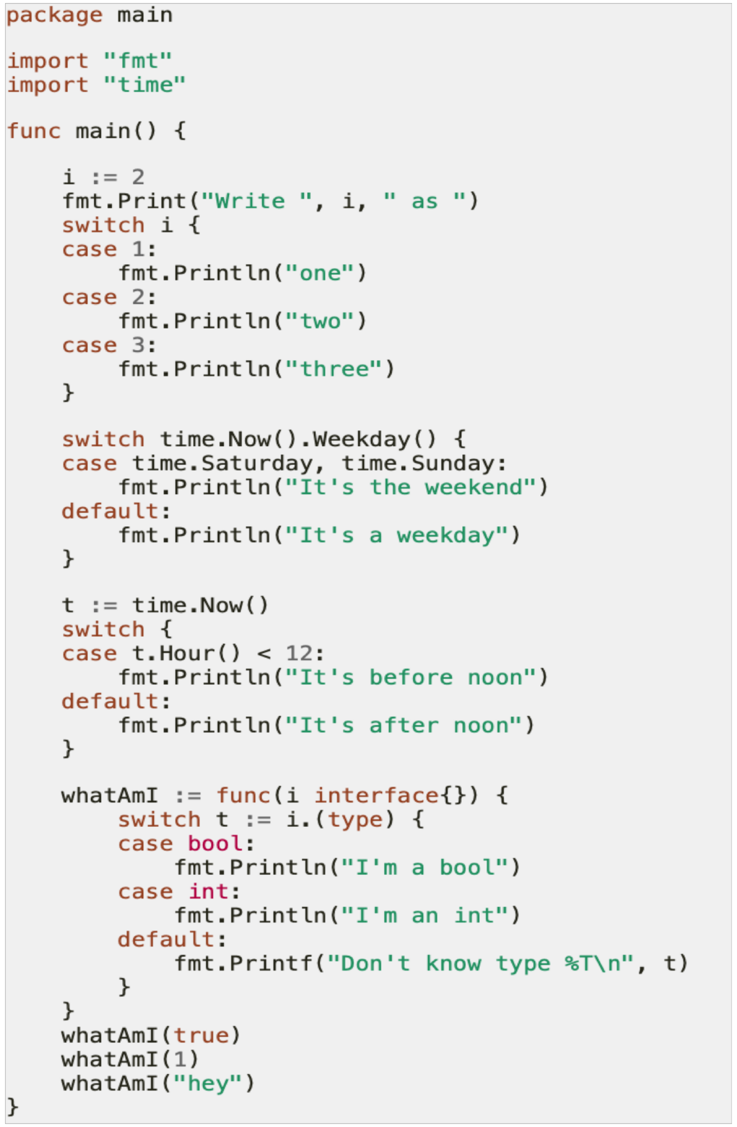
```go
package main

import "fmt"

func main() {

    if 7%2 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }

    if 8%4 == 0 {
        fmt.Println("8 is divisible by 4")
    }

    if num := 9; num < 0 {
        fmt.Println(num, "is negative")
    } else if num < 10 {
        fmt.Println(num, "has 1 digit")
    } else {
        fmt.Println(num, "has multiple digits")
    }
}
```

For Loops



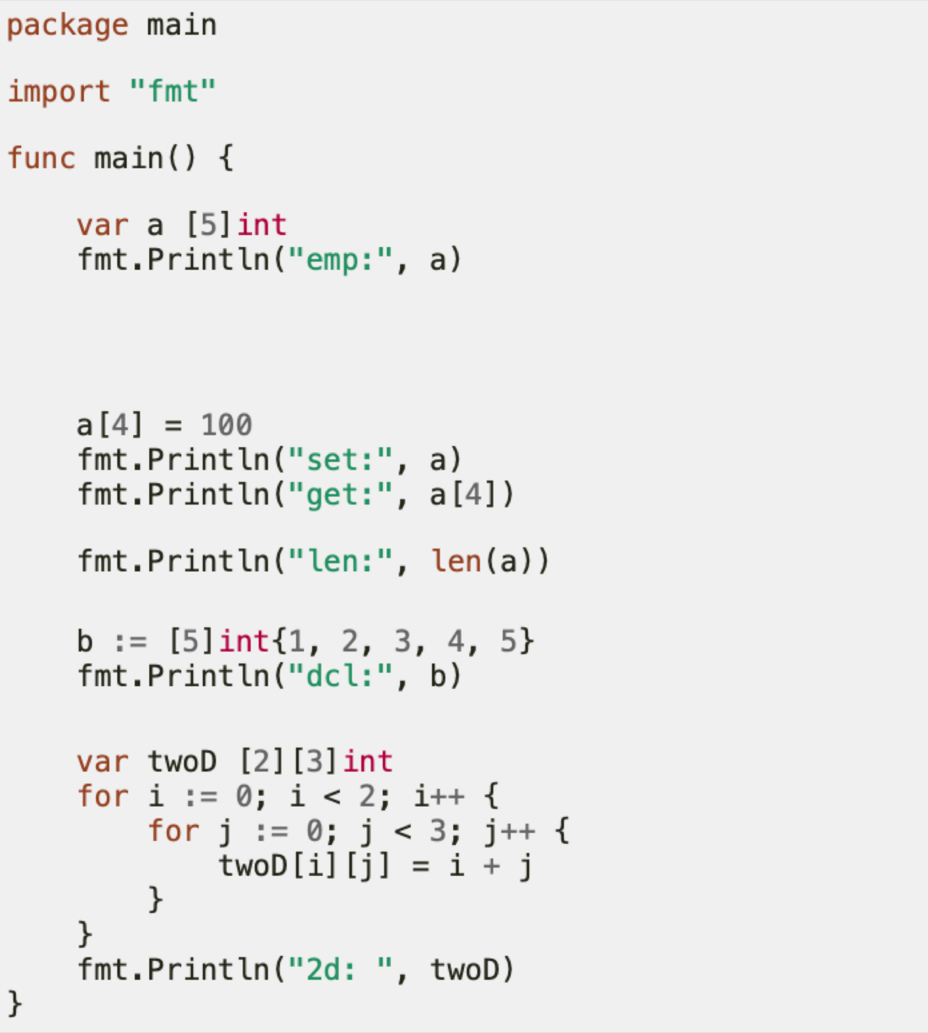
```go
package main

import "fmt"

func main() {

    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    for j := 7; j <= 9; j++ {
        fmt.Println(j)
    }

    for {
        fmt.Println("loop")
        break
    }

    for n := 0; n <= 5; n++ {
        if n%2 == 0 {
            continue
        }
        fmt.Println(n)
    }
}
```

Switch Statements



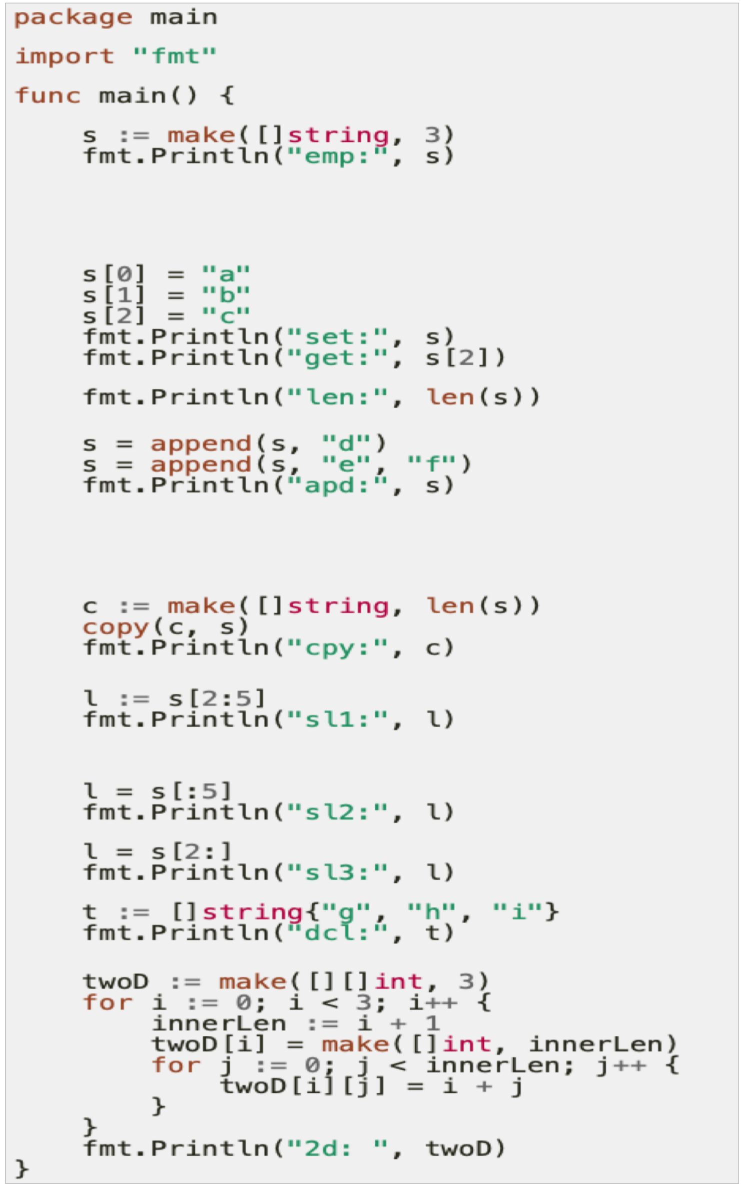
```go
package main

import "fmt"
import "time"

func main() {

    i := 2
    fmt.Print("Write ", i, " as ")
    switch i {
    case 1:
        fmt.Println("one")
    case 2:
        fmt.Println("two")
    case 3:
        fmt.Println("three")
    }

    switch time.Now().Weekday() {
    case time.Saturday, time.Sunday:
        fmt.Println("It's the weekend")
    default:
        fmt.Println("It's a weekday")
    }

    t := time.Now()
    switch {
    case t.Hour() < 12:
        fmt.Println("It's before noon")
    default:
        fmt.Println("It's after noon")
    }

    whatAmI := func(i interface{}) {
        switch t := i.(type) {
        case bool:
            fmt.Println("I'm a bool")
        case int:
            fmt.Println("I'm an int")
        default:
            fmt.Printf("Don't know type %T\n", t)
        }
    }
    whatAmI(true)
    whatAmI(1)
    whatAmI("hey")
}
```

Arrays



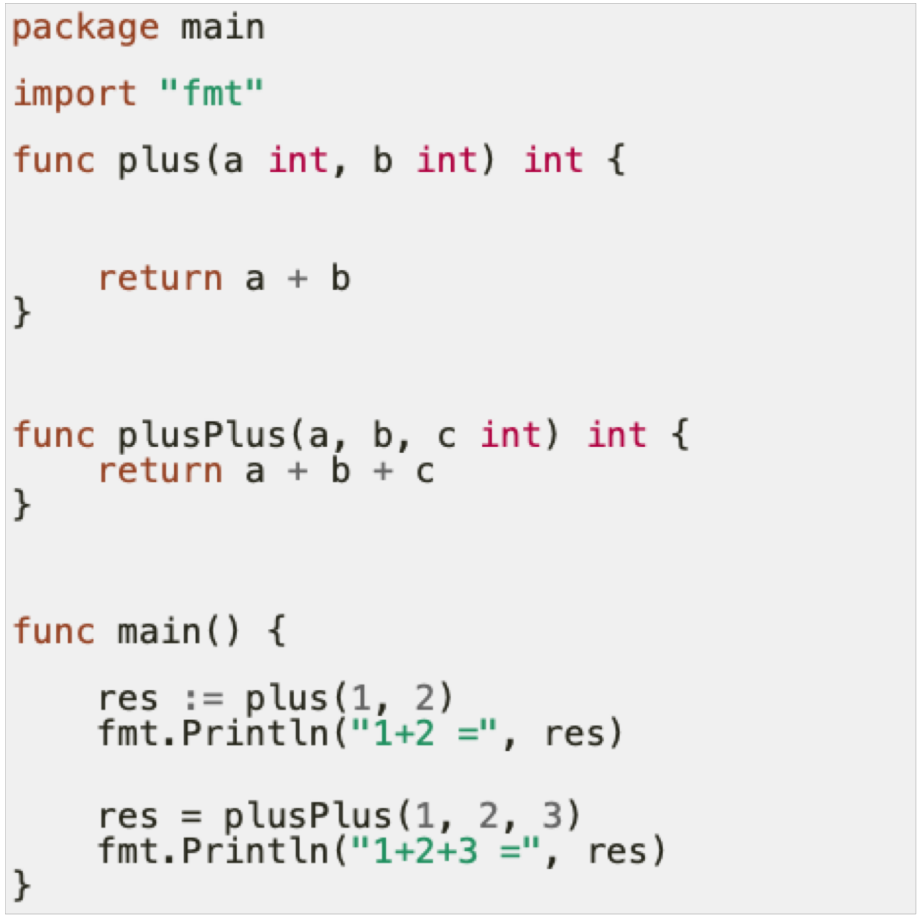
```
package main

import "fmt"

func main() {

    var a [5]int
    fmt.Println("emp:", a)




    a[4] = 100
    fmt.Println("set:", a)
    fmt.Println("get:", a[4])

    fmt.Println("len:", len(a))

    b := [5]int{1, 2, 3, 4, 5}
    fmt.Println("dcl:", b)

    var twoD [2][3]int
    for i := 0; i < 2; i++ {
        for j := 0; j < 3; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

Slices



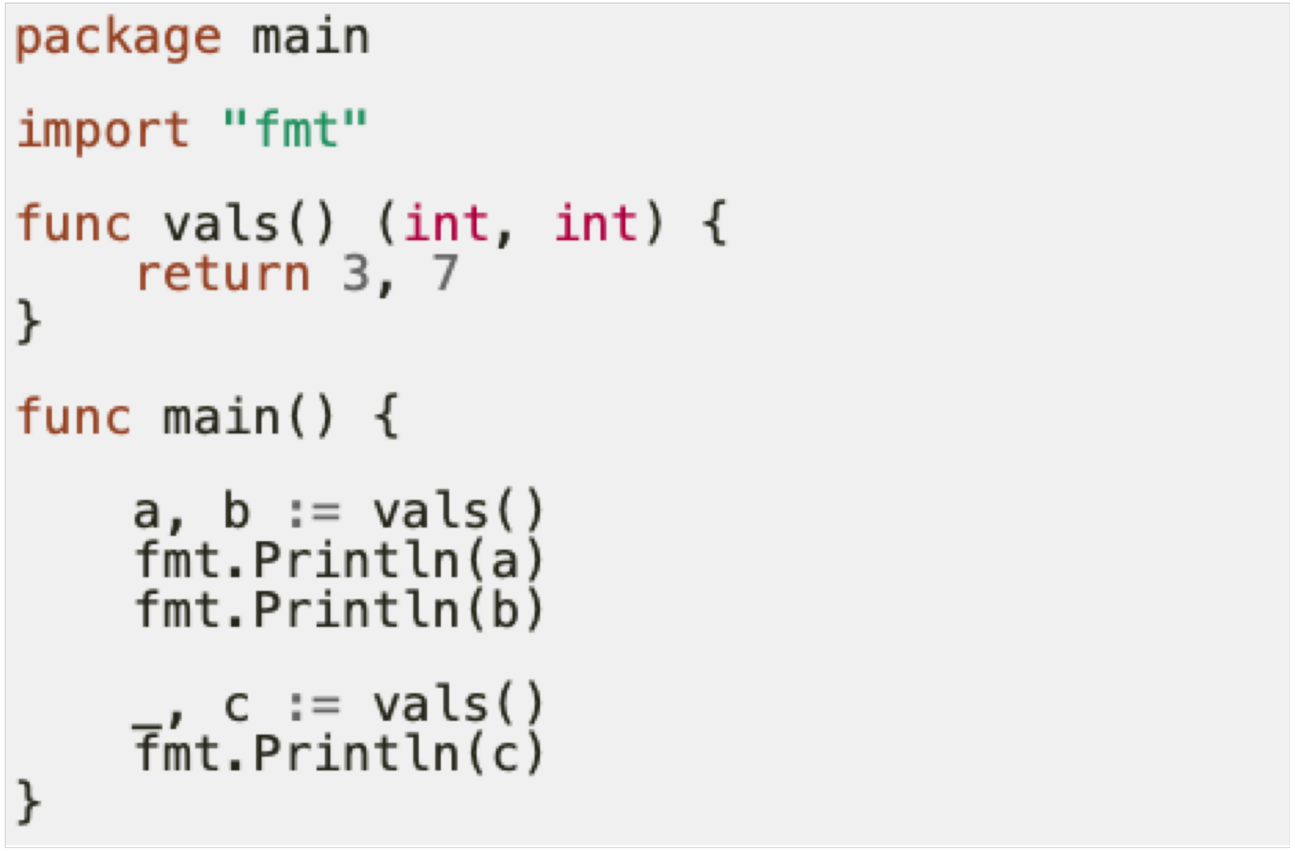
```go
package main

import "fmt"

func main() {

    s := make([]string, 3)
    fmt.Println("emp:", s)



    s[0] = "a"
    s[1] = "b"
    s[2] = "c"
    fmt.Println("set:", s)
    fmt.Println("get:", s[2])

    fmt.Println("len:", len(s))

    s = append(s, "d")
    s = append(s, "e", "f")
    fmt.Println("apd:", s)



    c := make([]string, len(s))
    copy(c, s)
    fmt.Println("cpy:", c)

    l := s[2:5]
    fmt.Println("sl1:", l)


    l = s[:5]
    fmt.Println("sl2:", l)

    l = s[2:]
    fmt.Println("sl3:", l)

    t := []string{"g", "h", "i"}
    fmt.Println("dcl:", t)

    twoD := make([][]int, 3)
    for i := 0; i < 3; i++ {
        innerLen := i + 1
        twoD[i] = make([]int, innerLen)
        for j := 0; j < innerLen; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2d: ", twoD)
}
```

Functions



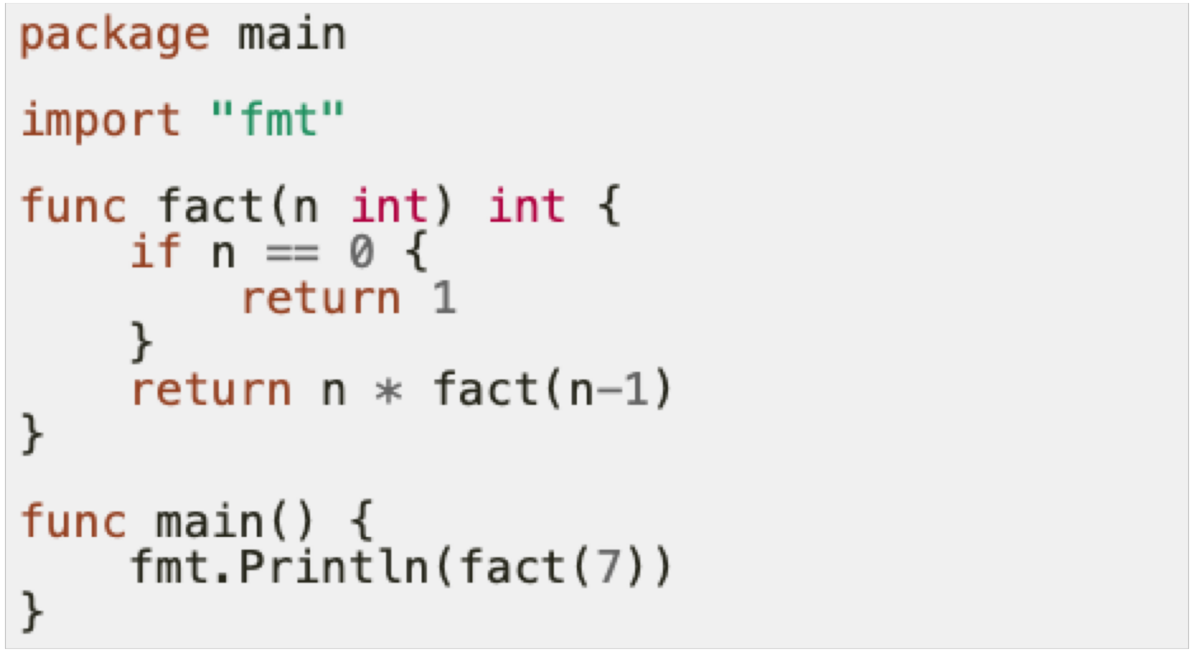
```
package main

import "fmt"

func plus(a int, b int) int {


    return a + b
}


func plusPlus(a, b, c int) int {
    return a + b + c
}


func main() {

    res := plus(1, 2)
    fmt.Println("1+2 =", res)

    res = plusPlus(1, 2, 3)
    fmt.Println("1+2+3 =", res)
}
```

Multiple Return Values



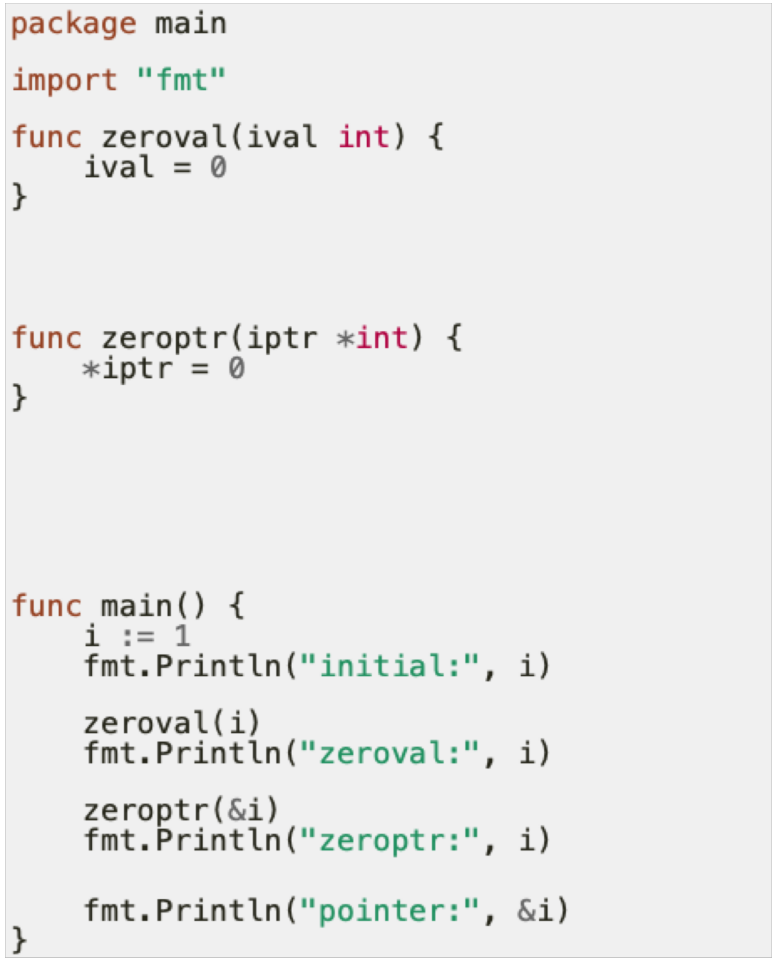
```go
package main

import "fmt"

func vals() (int, int) {
    return 3, 7
}

func main() {

    a, b := vals()
    fmt.Println(a)
    fmt.Println(b)

    _, c := vals()
    fmt.Println(c)
}
```

Recursion



```go
package main

import "fmt"

func fact(n int) int {
    if n == 0 {
        return 1
    }
    return n * fact(n-1)
}

func main() {
    fmt.Println(fact(7))
}
```

Pointers



```go
package main

import "fmt"

func zeroval(ival int) {
    ival = 0
}



func zeroptr(iptr *int) {
    *iptr = 0
}





func main() {
    i := 1
    fmt.Println("initial:", i)

    zeroval(i)
    fmt.Println("zeroval:", i)

    zeroptr(&i)
    fmt.Println("zeroptr:", i)

    fmt.Println("pointer:", &i)
}
```

Assignment 1

- Build a MapReduce library in Go to learn about fault tolerance in distributed systems
- 3 parts:
  - Part 1: Word Count
  - Part 2: Distributing MapReduce Jobs
  - Part 3: Handling Worker Failures

What is MapReduce?

- MapReduce is a distributed data processing paradigm
  - Introduced by Google in 2004
  - Popularized by the open-source Hadoop framework

- MapReduce dataflow:
  - Input reader
  - ***Map function***
  - Partition function
  - Comparison function
  - ***Reduce function***
  - Output Writer

Part 1: Word Count

- Read Section 2 of the MapReduce paper
- 2 functions to implement:
  - Map() is passed some text from a file. It should split it into words and return a list of key/value pairs
    - The key is a word and the value is the number of instances of this word in the line
  - Reduce() is called once for each key with a list of all the values for this key. It returns a single output value
    - Reduce takes input values for each word, sums them, and generates the final sum (i.e., how many word instances in the text)

Part 2: Distributing MapReduce Jobs

- Version of MapReduce that splits the work up over a set of workers
    - Master distributes jobs to workers through RPC calls

- Why would we like to do that?
    - Exploit multiple cores
    - Parallel execution
    - Performance speedup

- Goal: modify the RunMaster() function in master.go (mapreduce package)
    - Distribute jobs to workers
    - Return when all jobs are done

Part 2 (Cont’d)

- To get started, take a look at:
    - Run() in mapreduce.go:
        - Calls Split() to split the input into per-map-job files, then calls RunMaster() to run the map and reduce jobs, then calls Merge() to assemble the per-reduce-job outputs into a single output file
    - mr.registerChannel:
        - RPC handler passes the new worker's information through the channel to RunMaster, which should process new worker registrations by reading from this channel.
    - MapReduce struct, defined in mapreduce.go:
        - Stores information about the MapReduce job. You can modify the struct to keep track of additional state (if needed).

Part 3: Handling Worker Failures

- Workers might fail, master needs to re-assign jobs to an operational worker
  - RPCs issued by master to a worker that failed timeout
- ***Assumption:*** worker may never fail while executing a job
- ***Assumption:*** the master does not fail

Good luck with assignment 1!