Go Tutorial

To do …

- Today: A brief, gentle intro to Go
- Next: An overview of remote invocation paradigms (request-reply, RPC and RMI)
About Go

- Developed by Google
  - Webpage: https://golang.org/

- Concurrency was a priority in the language design

- A bit of a mix between high- and low-level programming language features
Go Features

- Some similarities with C
  - Looks like it
  - Compiled
  - Fixed-size int, float, and complex types
  - Strong, static typing
  - Uses return codes instead of exceptions
  - No classes/inheritance (uses structs/interfaces)
  - Has pointers
Go Features

- And a bit of Python
  - No pointer arithmetic
  - Has garbage collection
  - Key, value maps are part of the language
  - Can return multiple values (useful for error codes)
  - Lots of “batteries-included” libraries
    - flags, ginkgo (testing), logrus (logs), net/http, net/rpc, encoding/json, encoding/gob
Go Features

- Well-suited for designing concurrent, distributed systems
  - goroutines
  - channels
  - defer
  - RPC library

- (more on these later)
Installing Go (it’s easy)

- Available in most package managers
  - Macports & Homebrew: install “go”
  - Ubuntu & Fedora: “golang”
  - Arch: “go”

- Binary distributions available for most OSes
  - https://golang.org/doc/install

- I’ll be using the latest version of Go to grade
  - T-Lab and Wilkinson should be up-to-date
Hello, 世界

```go
// This is a comment

package main
// Programs start running in package "main"

import "fmt"

func main() {
    // strings support Unicode!
    fmt.Println("Hello, EECS 345! (あいうえお)")
}
```

Hello, EECS 345! (あいうえお)

Program exited.

http://play.golang.org/p/pL_36Jnu7m
Go’s “net” package uses byte slices ([]byte)
  – Can convert string to []byte
    • []byte(str)
  – And []bytes to string
    • string(byte_slice)
package main

import "fmt"

func main() {
    // the type goes last!
    var i int
    // or is implied with a colon
    j := 1
    fmt.Println(i, j)
}

0 1

Program exited.
More declaring and types

```go
package main

import "fmt"

func main() {
    // be careful about types!
    k := 3    // is an int!
    j := 3.0  // is a float!
    fmt.Println(k, j, float64(k)+j)
}
```

3 3 6

Program exited.

http://play.golang.org/p/hW_3xJp7dF
package main

import "fmt"

func swap(x, y string) (string, string) {
    return y, x
}

func main() {
    a, b := swap("hello", "world")
    fmt.Println(a, b)
}
package main

import "fmt"

func split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}

func main() {
    fmt.Println(split(17))
}

Program exited.

http://play.golang.org/p/7VpZG0mfG7
package main

import "fmt"

func main() {
    sum := 1
    for sum < 1000 {
        sum += sum
    }
    fmt.Println(sum)
}

http://play.golang.org/p/8c5bGBbLcq

There can be only one (looping construct)

1024

Program exited.
package main

import "fmt"

func main() {
    x := 0
    if v := 1; v > x {
        fmt.Println(v,"is greater than",x)
    }
}

1 is greater than 0

Scope is limited to the “if” and “else” clauses statement

Program exited.
package main

import "fmt"

func main() {
    defer fmt.Println("world")
    fmt.Println("hello")
}

hello
world

Great for things like closing files or connections!

Program exited.

http://play.golang.org/p/tYcMMTOwFa
package main

import (
    "fmt"
    "math"
)

type Vertex struct {
    X, Y float64
}

func (v *Vertex) Abs() float64 {
}

func main() {
    v := Vertex{3, 4}
    fmt.Println(v.Abs())
}
Interfaces

- An interface type is basically a set of required methods.

- Any type (struct) that implements the required methods, implements that interface.

- A type is not explicitly declared to be of a certain interface, it is implicit.
  - Just implement the required methods.
Interface example

```go
1 type Abser interface {
2     Abs() float64
3 }

4 type Vertex struct {
5     X, Y float64
6 }

7 func (v *Vertex) Abs() float64 {
9 }
```

Type Vertex meets the requirements of Abser interface

http://play.golang.org/p/KbyhiM29tQ
package main

import (
    "fmt"
    "strconv"
)

func main() {
    i, err := strconv.Atoi("42")
    if err != nil {
        fmt.Printf("couldn't convert: %v\n", err)
    }
    fmt.Println("Converted integer: ", i)
}
package main

import (
    "net"
)

func handleConnection(conn net.Conn) {
    // do something
}

func main() {
    ln, err := net.Listen("tcp", ":8080")
    if err != nil {
        // handle error
    }
    for {
        conn, err := ln.Accept()
        if err != nil {
            // handle error
        }
        handleConnection(conn)
    }
}
package main
import (
    "net"
)

func handleConnection(conn net.Conn) {
    // do something
}

func main() {
    ln, err := net.Listen("tcp", ":8080")
    if err != nil {
        // handle error
    }
    for {
        conn, err := ln.Accept()
        if err != nil {
            // handle error
        }
        go handleConnection(conn)
    }
}
Concurrency and shared memory

- **Locks are a pain**
  - Must have global lock ordering
  - Error-prone
  - Extremely difficult to debug

- **Message passing**
  - Have to explicitly say what you’re sharing
  - But easier to maintain in large, complicated programs
  - Go uses “channels” to pass data
package main

import "fmt"

func fib(c chan int) {
    i, j := 0, 1
    for {
        i, j = i + j, i
        c <- i
    }
}

func main() {
    c := make(chan int)
    go fib(c)
    for i := 0; i < 10; i++{
        fmt.Println(<-c)
    }
}

http://play.golang.org/p/ozAf6hJXQt
By default, channel operations block

Buffered channels do not block if they are not full

```go
1 c := make(chan int, 1) // (type, buff_size)
2 c <- 1 // will not block

// blocks if the channel is full
// (another goroutine has not yet read
// from the channel)
5 c <- 2
```
Using select

```go
func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
            case c <- x:
                x, y = y, x+y
            case <-quit:
                fmt.Println("quit")
                return
        }
    }
}
```

http://play.golang.org/p/e8xP0_99S0
Back to channels

- Select statement also does non-blocking channel I/O

```
Reading

1  messages := make(chan string)
2  select {
3    case msg := <-messages:
    fmt.Println("received message", msg)
3    default:
4      fmt.Println("no message received")
5  }

Writing

1  select {
2    case messages <- msg:
3      fmt.Println("sent message", msg)
4    default:
5      fmt.Println("no message sent")
6  }
```
A few more things...

- Go can be somewhat picky
  - Unused variables raise errors, not warnings
    - Use "_" for variables you don’t care about
  - Unused imports raise errors
    - "goimports" is a project to automatically add/remove imports (use at your own risk)
  - "go fmt" can auto-indent your code
  - } else if {
    - "} else" must be on the same line