Golang

•••

Tiffany Tillett
Hector Acosta

Overview

- Developed at google
- Emphasis on simplicity
- Statically typed
  - Type inference (e.g. x := 42 vs int x = 0);
- Fast compilation times
- Package management built into the language
- Statically linked (by default)
- Interfaces (not to be confused with inheritance)
  - No generics (yet)
- **Concurrency primitives**

Concurrency vs Parallelism

Concurrency: Programming as the composition of independently executing processes.

Parallelism: Programming as the simultaneous execution of (possibly related) computations.

Concurrency Constructs

- Channels
- Goroutines
- Select Statements
- Fan-in
- Slices

Goroutines

A goroutine is a function that is capable of running concurrently with other functions. To create a goroutine we use the keyword go followed by a function invocation.

Goroutines

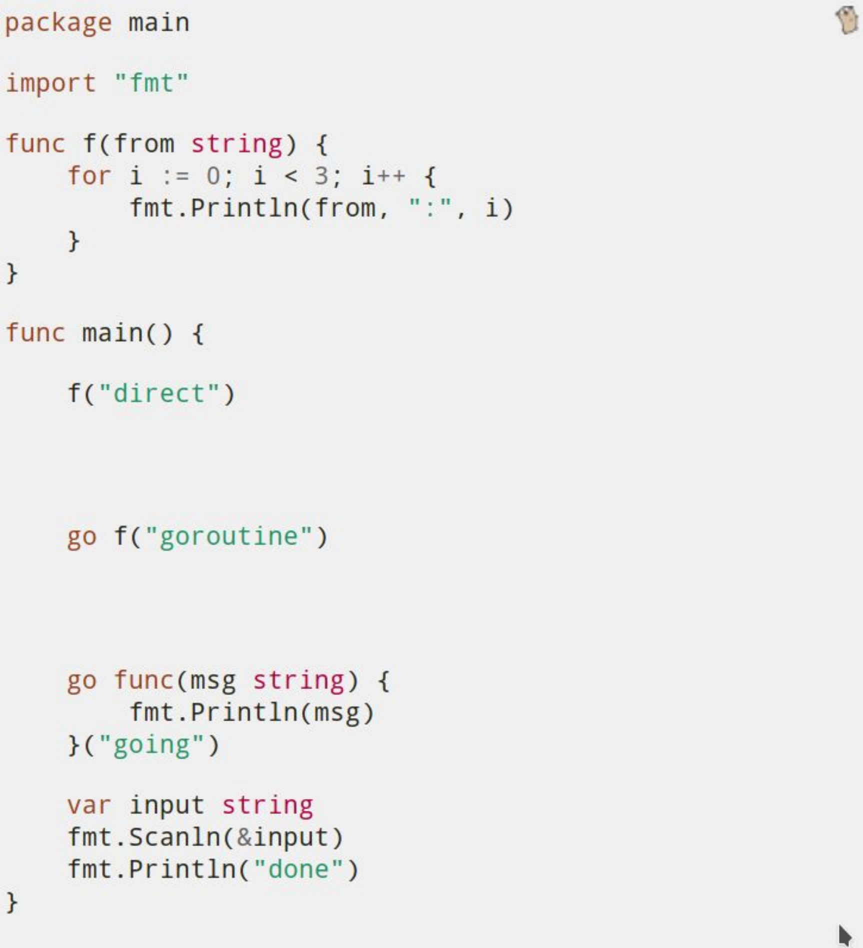
Think threads (but not really)

Lightweight thread of execution.

&

Typed values that allow goroutines to

exchange data



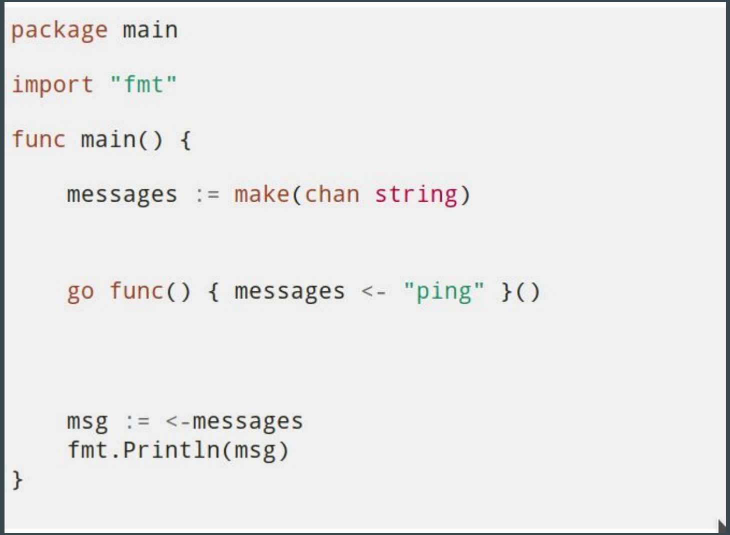
```go
package main

import "fmt"

func f(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":", i)
    }
}

func main() {

    f("direct")



    go f("goroutine")



    go func(msg string) {
        fmt.Println(msg)
    }("going")

    var input string
    fmt.Scanln(&input)
    fmt.Println("done")
}
```

Channels

Typed values that allow goroutines to exchange data

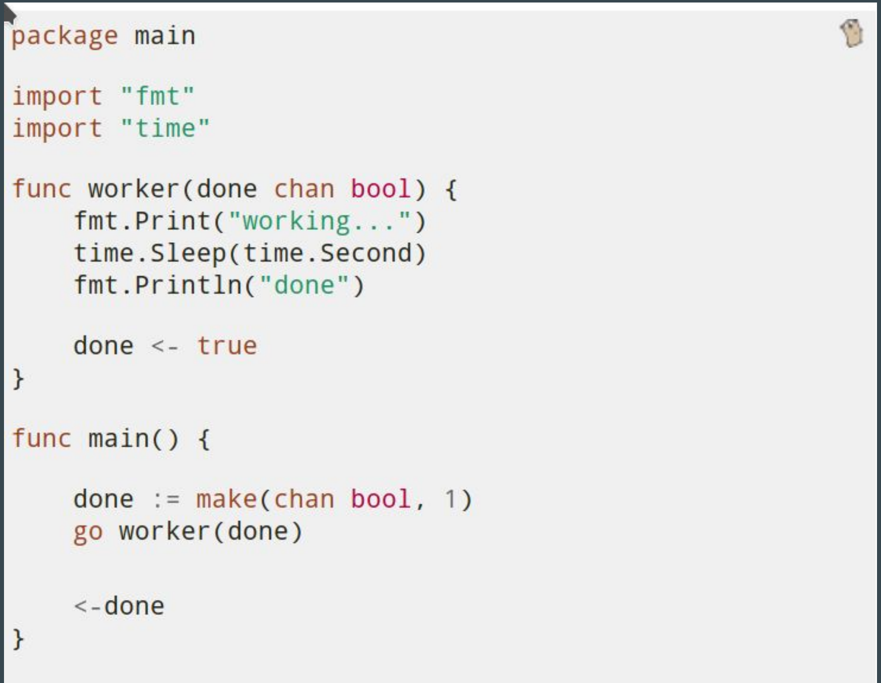
Create a new channel with make(chan val-type).

Buffered, blocking



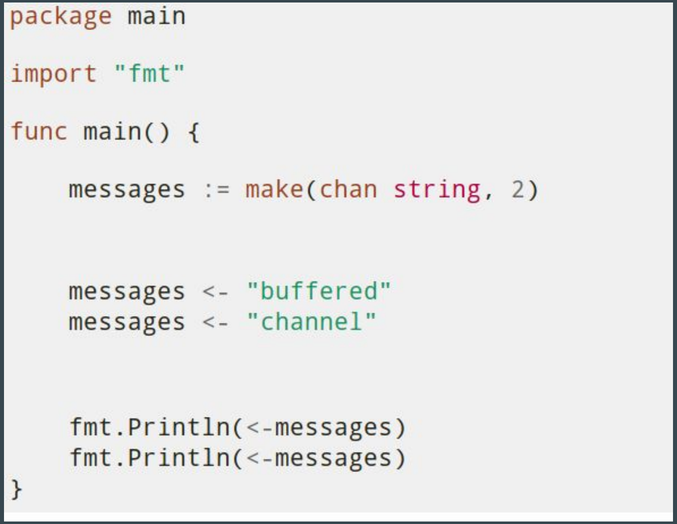
```go
package main

import "fmt"

func main() {

    messages := make(chan string)


    go func() { messages <- "ping" }()



    msg := <-messages
    fmt.Println(msg)
}
```

Channels Cont.

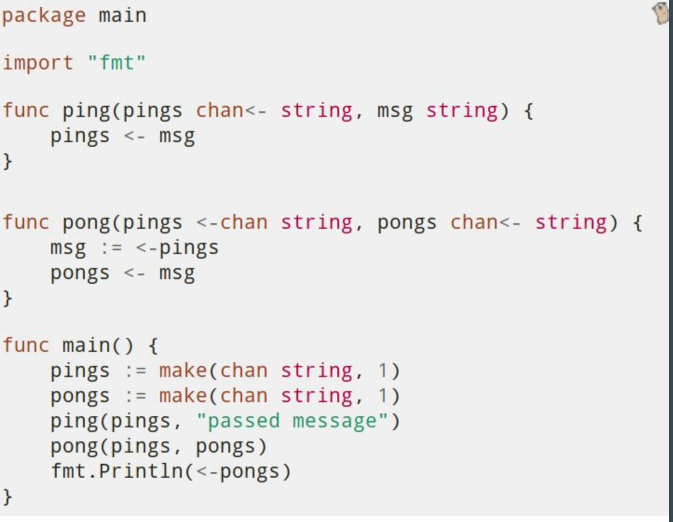


```go
package main

import "fmt"
import "time"

func worker(done chan bool) {
    fmt.Print("working...")
    time.Sleep(time.Second)
    fmt.Println("done")

    done <- true
}

func main() {

    done := make(chan bool, 1)
    go worker(done)


    <-done
}
```

Channels Cont.



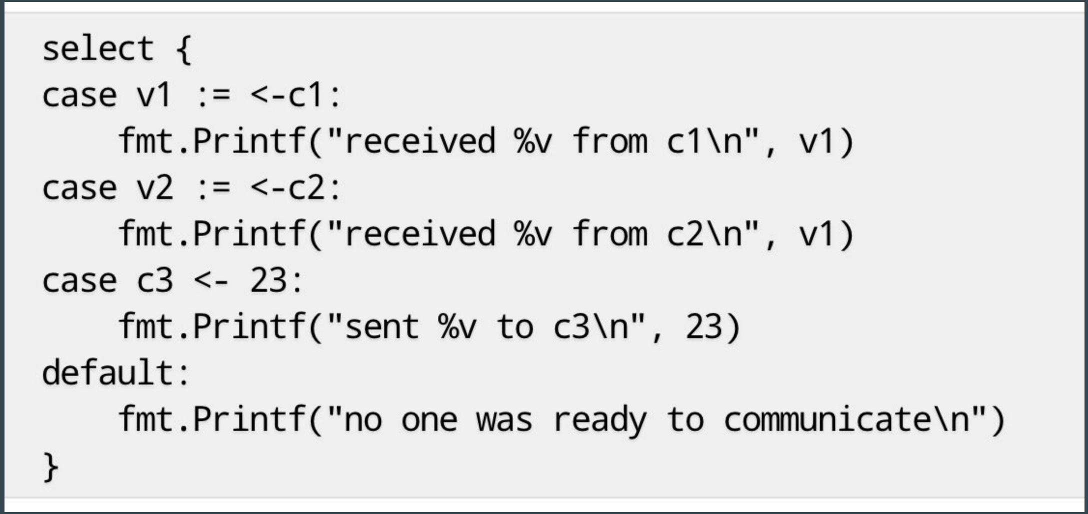
```go
package main

import "fmt"

func main() {

    messages := make(chan string, 2)


    messages <- "buffered"
    messages <- "channel"


    fmt.Println(<-messages)
    fmt.Println(<-messages)
}
```



```go
package main

import "fmt"

func ping(pings chan<- string, msg string) {
    pings <- msg
}

func pong(pings <-chan string, pongs chan<- string) {
    msg := <-pings
    pongs <- msg
}

func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)
    ping(pings, "passed message")
    pong(pings, pongs)
    fmt.Println(<-pongs)
}
```

Select

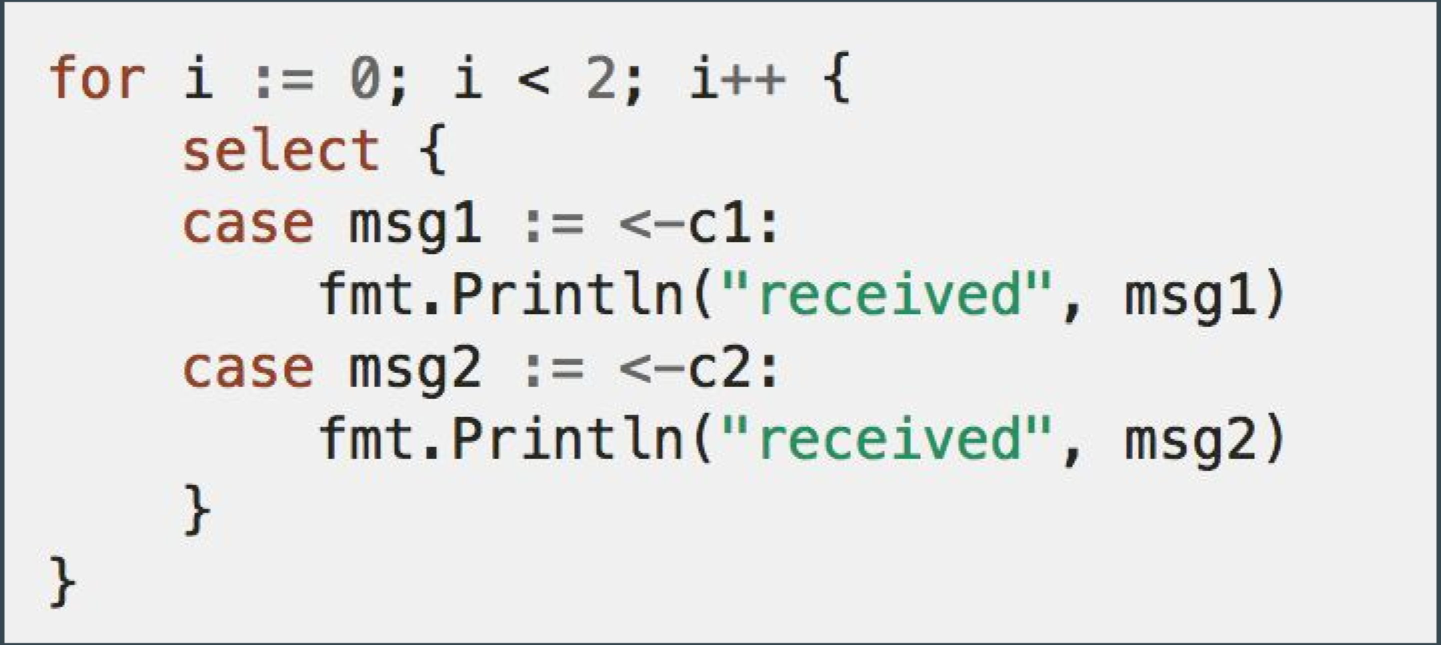
- Select statements are a way to read from/write to multiple channels
- It will check to see which channel is ready to send/receive data. If multiple are ready, it arbitrarily chooses.
- If none are ready, it will block until a channel is ready



```
select {
case v1 := <-c1:
    fmt.Printf("received %v from c1\n", v1)
case v2 := <-c2:
    fmt.Printf("received %v from c2\n", v1)
case c3 <- 23:
    fmt.Printf("sent %v to c3\n", 23)
default:
    fmt.Printf("no one was ready to communicate\n")
}
```

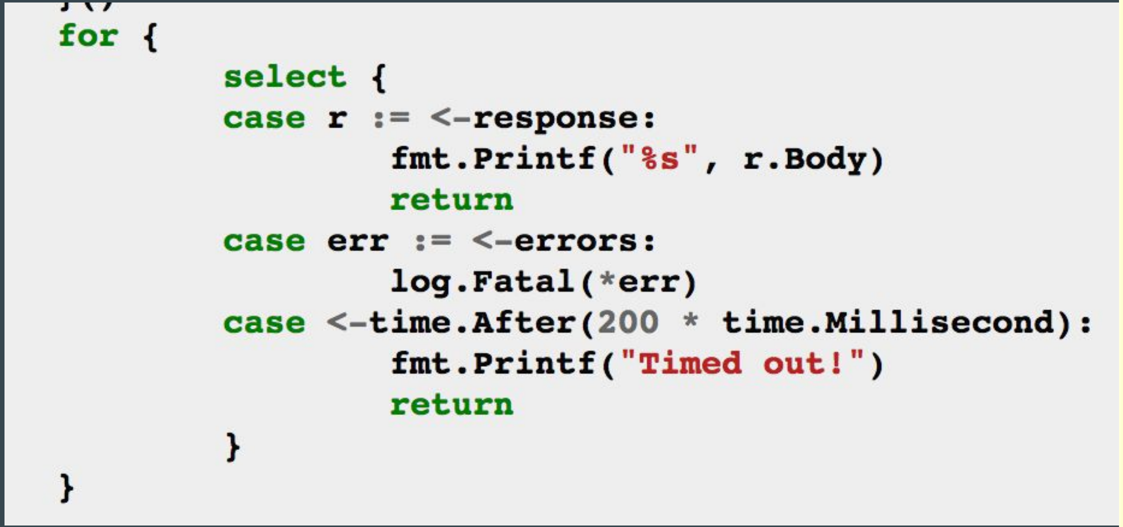
Select Cont.

- Select statements be used inside of loops

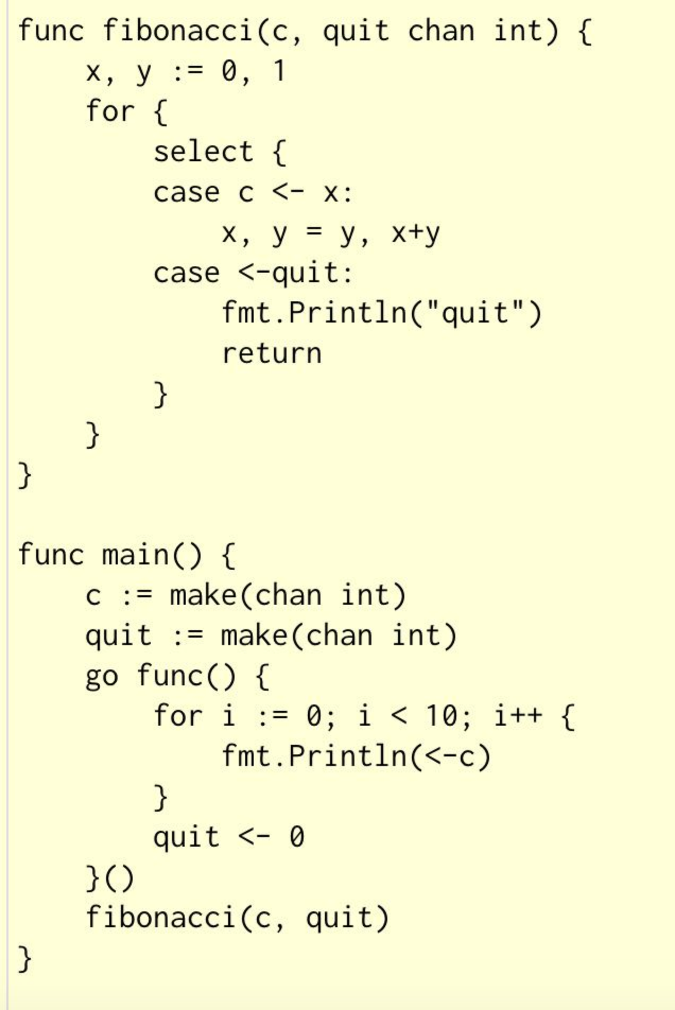


```go
for i := 0; i < 2; i++ {
    select {
    case msg1 := <-c1:
        fmt.Println("received", msg1)
    case msg2 := <-c2:
        fmt.Println("received", msg2)
    }
}
```

Select Cont.



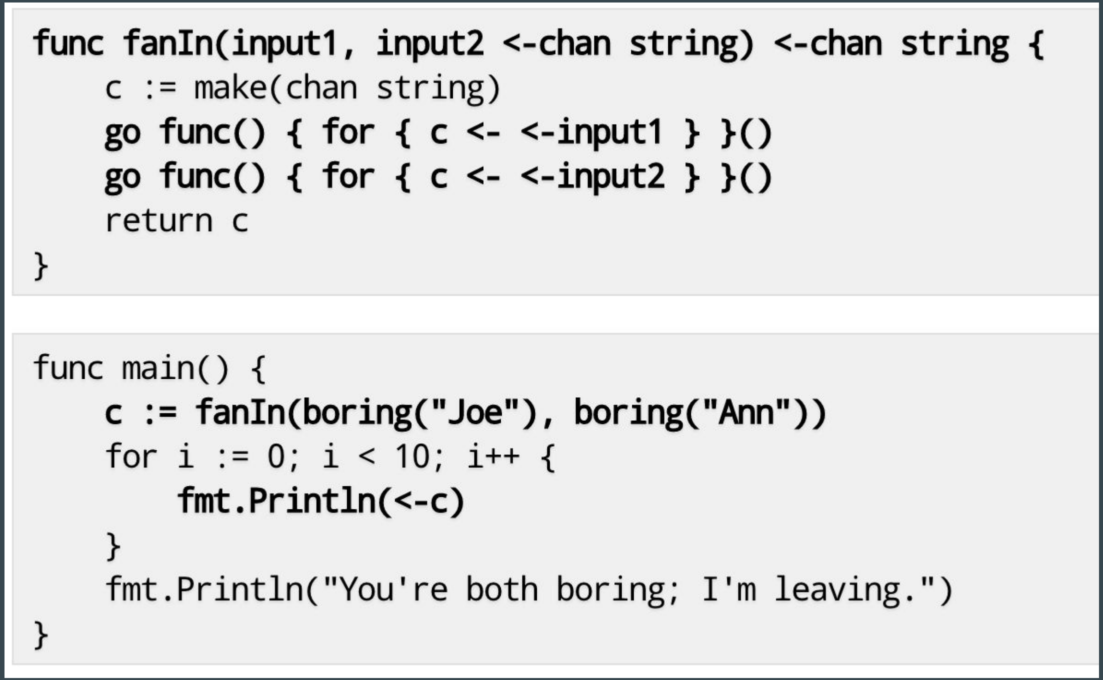
```
}()
for {
        select {
        case r := <-response:
                fmt.Printf("%s", r.Body)
                return
        case err := <-errors:
                log.Fatal(*err)
        case <-time.After(200 * time.Millisecond):
                fmt.Printf("Timed out!")
                return
        }
}
```



```
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
        }
    }
}

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

Fan-In

- A Fan-In can combine results from multiple channels into a single channel

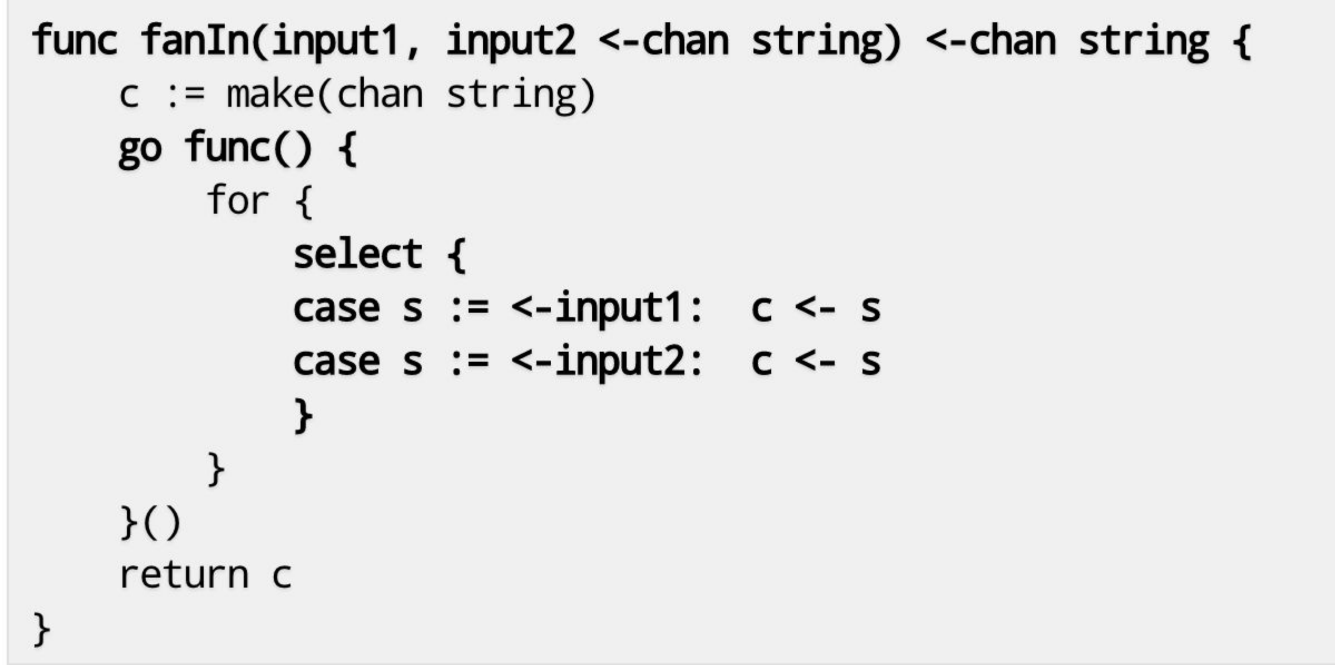


```
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() { for { c <- <-input1 } }()
    go func() { for { c <- <-input2 } }()
    return c
}
```

```
func main() {
    c := fanIn(boring("Joe"), boring("Ann"))
    for i := 0; i < 10; i++ {
        fmt.Println(<-c)
    }
    fmt.Println("You're both boring; I'm leaving.")
}
```

Fan-In with Select

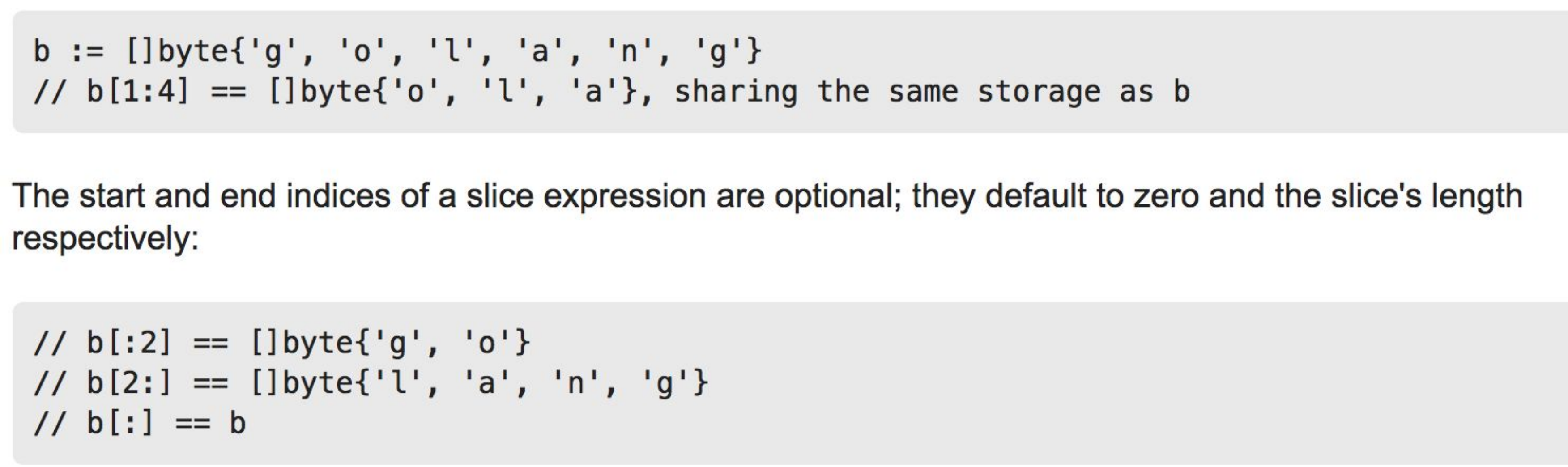
- We can combine fan-in with select to reduce the number of goroutines required



```go
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() {
        for {
            select {
            case s := <-input1:  c <- s
            case s := <-input2:  c <- s
            }
        }
    }()
    return c
}
```

Slices

- Slices are a way to select a subset of an array or another slice
- Start point is inclusive; end point is exclusive



```
b := []byte{'g', 'o', 'l', 'a', 'n', 'g'}
// b[1:4] == []byte{'o', 'l', 'a'}, sharing the same storage as b
```

The start and end indices of a slice expression are optional; they default to zero and the slice's length respectively:

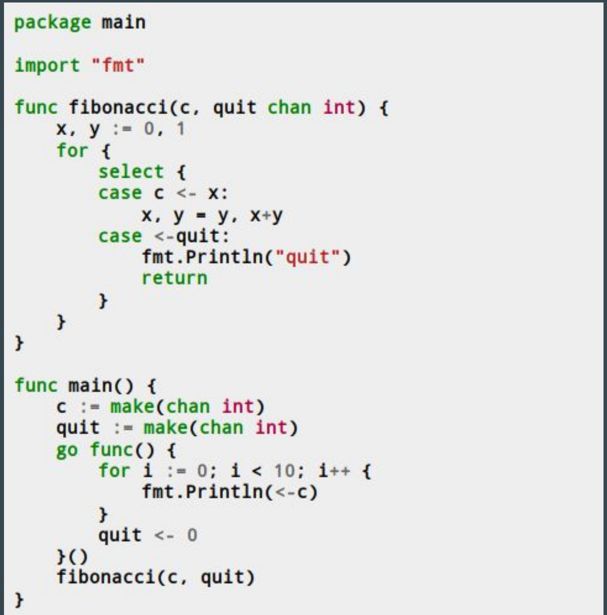
```
// b[:2] == []byte{'g', 'o'}
// b[2:] == []byte{'l', 'a', 'n', 'g'}
// b[:] == b
```

Examples

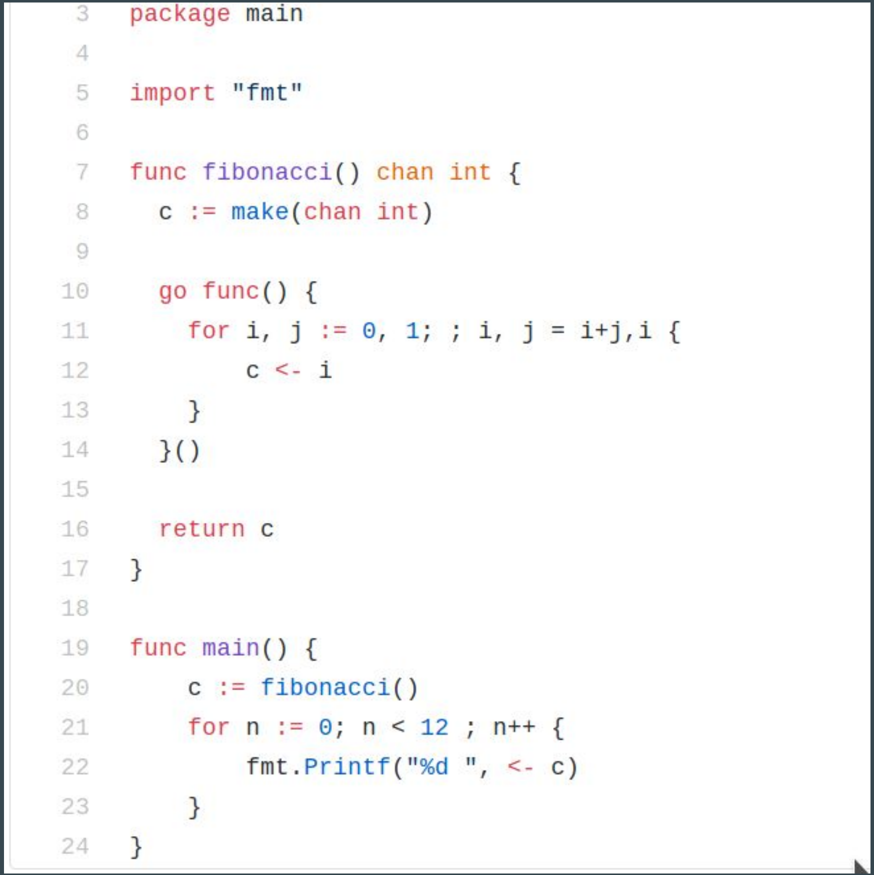
Concurrent fibonacci

Reduce

Concurrent Fibonacci

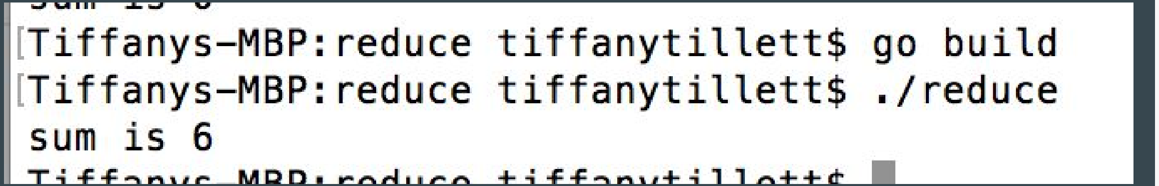


```go
package main

import "fmt"

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
        }
    }
}

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

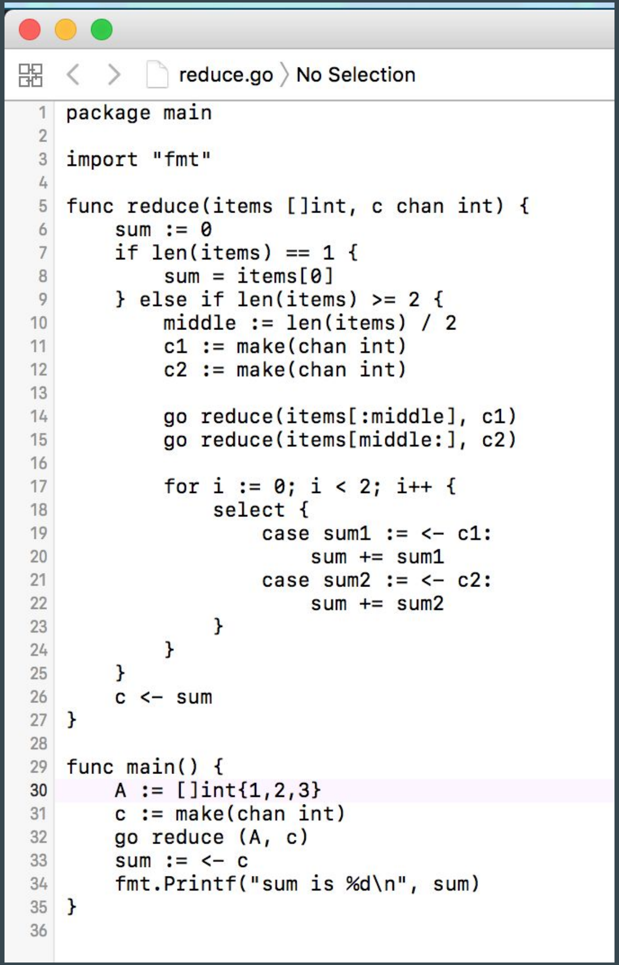


```go
package main

import "fmt"

func fibonacci() chan int {
  c := make(chan int)

  go func() {
    for i, j := 0, 1; ; i, j = i+j,i {
        c <- i
    }
  }()

  return c
}

func main() {
    c := fibonacci()
    for n := 0; n < 12 ; n++ {
        fmt.Printf("%d ", <- c)
    }
}
```

Reduce



```
sum is 6
Tiffanys-MBP:reduce tiffanytillett$ go build
Tiffanys-MBP:reduce tiffanytillett$ ./reduce
sum is 6
Tiffanys-MBP:reduce tiffanytillett$ 
```



```go
package main

import "fmt"

func reduce(items []int, c chan int) {
    sum := 0
    if len(items) == 1 {
        sum = items[0]
    } else if len(items) >= 2 {
        middle := len(items) / 2
        c1 := make(chan int)
        c2 := make(chan int)

        go reduce(items[:middle], c1)
        go reduce(items[middle:], c2)

        for i := 0; i < 2; i++ {
            select {
                case sum1 := <- c1:
                    sum += sum1
                case sum2 := <- c2:
                    sum += sum2
            }
        }
    }
    c <- sum
}

func main() {
    A := []int{1,2,3}
    c := make(chan int)
    go reduce (A, c)
    sum := <- c
    fmt.Printf("sum is %d\n", sum)
}
```

Resources

- Go Language
  - https://talks.golang.org/2012/concurrency.slide#1
  - http://www.golangbootcamp.com/book/concurrency
  - https://appliedgo.net/generics/
  - https://golang.org/pkg/
  - https://golang.org/doc/
- Installation and Running
  - https://golang.org/dl/