



Go in a Nutshell

Alexandre David

1.2.05

adavid@cs.aau.dk





Introduction

- Go is a script-like, object-like, typed language.
 - It is compiled and runs efficiently.
 - It has a garbage collector.
 - It supports concurrency.



Hello World

- Define & import packages.
- Functions defined with 'func' keyword.
- Source file in UTF-8.
- No semi-colon.

```
package main  
import fmt "fmt"
```

```
func main() {  
    fmt.Printf("Hello world\n")  
}
```



How to compile

- 32-bit:

- 8g helloworld.go
8l helloworld.8
8.out

- 64-bit:

- 6g helloworld.go
6l helloworld.6
6.out



Variables

- Constants: `const name = value.`
 - Integers constants may be big, e.g.,
`const big = (1 << 100) >> 97`
They overflow only when assigned to variables.
- Variables: `var name = initValue.`
- `var omitNewLine = flag.Bool("n",false,"no new line")`
 - make a `Bool` defined in package `flag`,
 - has type `*Bool` (pointer to a `Bool`).



Using Packages

- `fmt.Printf`
 - package `fmt`, function `Printf`.
- `os.Stdout.WriteString(s)`
 - package `os`, variable `Stdout`, method `WriteString`.



Common types

- int float
- int8, int32, float64...
- string (immutable)
- Arrays:
 - *var name [size]type;*
var arrayOfInt[10]int;
 - Slice expressions *a[low:high]* represent sub-arrays.
 - Multiple slices can share data, multiple arrays cannot.



Slice - Example

```
func sum(a []int) int { // returns an int
    s := 0
    for i := 0; i < len(a); i++ {
        s += a[i]
    }
    return s
}
```

```
s := sum(&[3]int{1,2,3}) // &->address of instance
                        // implicitly promoted to a slice
s := sum(&[...]int{1,2,3}) // let the compiler count
s := sum([]int{1,2,3})    // also works
```




cont.

- Maps can be initialized in a similar manner:
`m := map[string]int{"one":1, "two":2}`
- Ranges are supported:
 - `for i:=0; i<len(a); i++ {...}`
→
`for i, v := range a {...}`
assigns `i` to the indices and `v` to values of `a`.



Allocation

- `type T struct { a,b int }`
`var t *T = new(T)`
or
`t = new(T)`
 - `new()` allocates and constructs a new instance.
- To construct without getting a pointer use `make`:
 - `m := make(map[string]int)`



Assignments

- Some functions return pairs, typically value and error status.
 - `nr,er := f.Read(&buf)`



Switch

- General switch statements:

```
switch init; value {  
  case expr :
```

```
    ...
```

```
  case expr :
```

```
    ...
```

```
}
```

- Matches from top to bottom expr to value.
- Has an init statement.

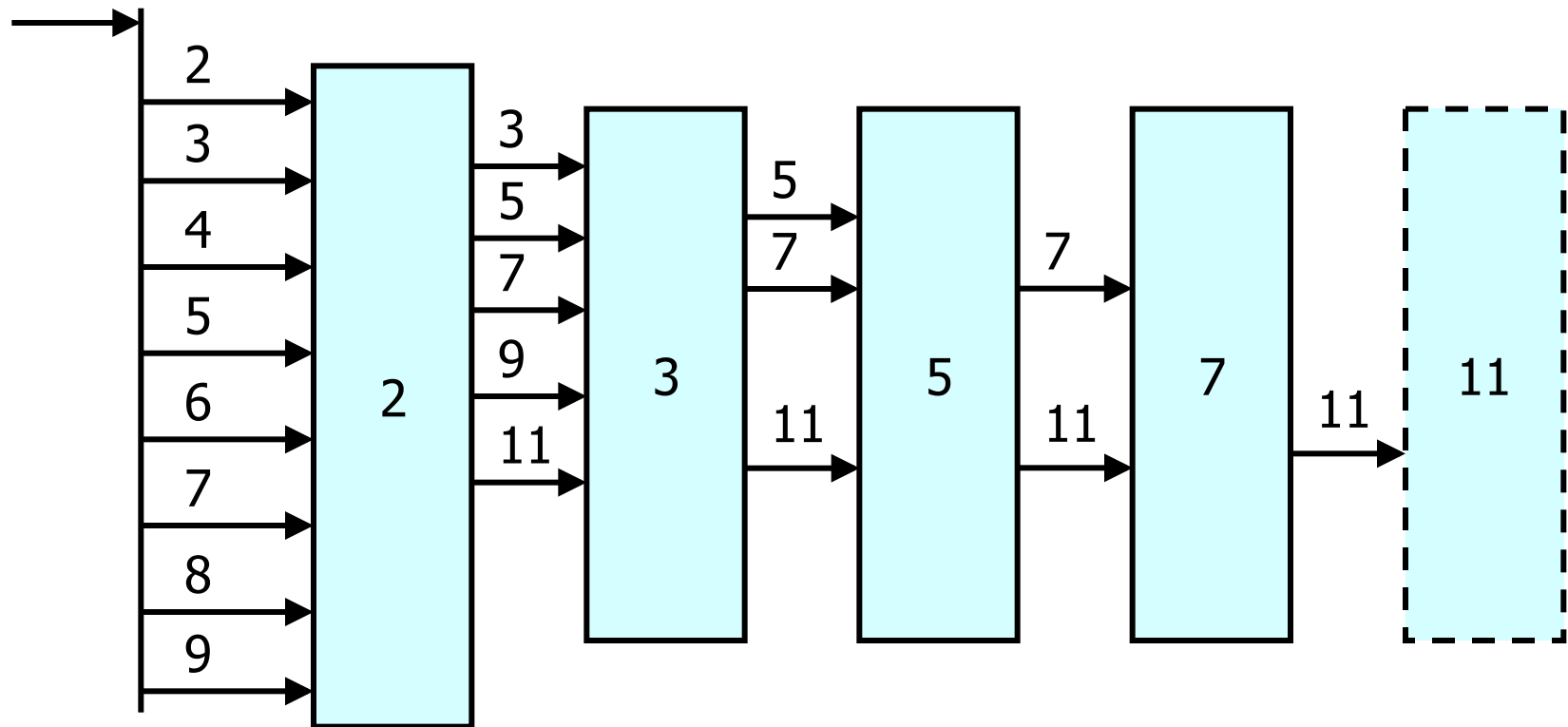


Interfaces

- type reader interface {
 Read(b []byte) (ret int, err os.Error)
 String() string
}
- Any type having these methods implements the interface.
- Separate implementation from API.

Concurrency – Prime Numbers

- Based on Sieve of Eratosthenes.
- Put filters to filter streams of numbers.





Channels

- Use Go channels to have processes communicate with each other.
 - ```
func generate(ch chan int) {
 for i:=2; ; i++ {
 ch <- i // Send 'i' to channel 'ch'
 }
}
```
  - Channels are blocking.
  - Operator <- sends values to channels.
  - Simple form of typed message passing.
  - Can be sent as part of messages.



# Filters

- Copy values from 'in' to 'out', discarding multiples of 'prime'.
  - ```
func filter(in,out chan int, prime int) {  
    for {  
        i := <-in           // Receives value from  
        'in'  
        if i % prime != 0 {  
            out <- i         // Send value to  
            'out'  
        }  
    }  
}
```
 - Concurrent computations are called *goroutines*.
 - The filters and generators are goroutines.



All together

- ```
func main() {
 ch := make(chan int) // create channel
 go generate(ch) // start goroutine
 for {
 prime := <- ch
 fmt.Println(prime)
 ch1 := make(chan int)
 go filter(ch, ch1, prime)
 ch = ch1
 }
}
```
- Create and start filters on-the-fly & connect them.



# Lambda Expressions

---

- Inlined functions.

- ```
func filter(in chan int, prime int) chan int {  
    out := make(chan int)  
    go func() {  
        for {  
            if i := <-in; i%prime != 0 {  
                out <- i  
            }  
        }  
    }()  
    return out  
}
```



Conclusion

- Light weight language.
- Object like.
- Support for concurrency
 - goroutines
 - message passing via (blocking) channels