The Go Programming Language

Fred Mora

Why another language?

- You can't shake a mouse these days without hitting a new language

- Scripting languages are a dime a dozen. Good ones are rare.
  - PHP? The 70s called, they want their patchwork quilt back!
  - Python? Nice of you to keep backwards compatibility... NOT.
- Compiled languages are much less common
  - C is getting long in the teeth
  - C++ is to practical languages what a Dali elephant is to a horse.

What is Go?

- Go is impossible to search. Synonym: Golang

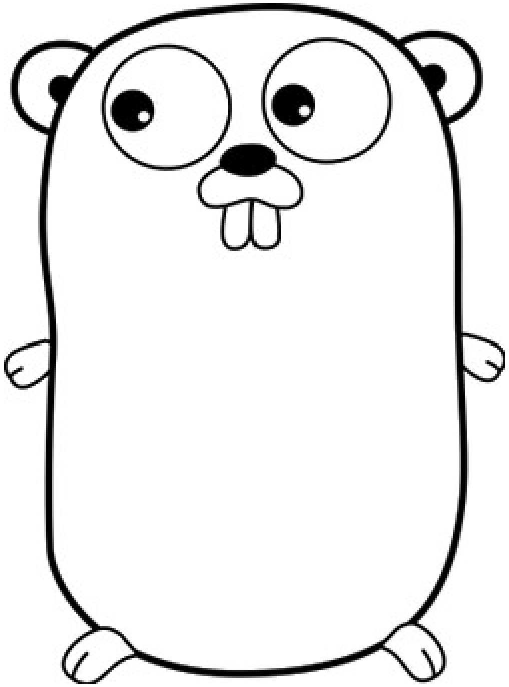
- Named after the Japanese game

- Open source, BSD-style license

- Sponsored by Google

- Version 1 released in 2012, now at v 1.5. Backwards compatible incremental changes!

- Design goals:
  - Viable for systems programming
  - Easy tool chain integration

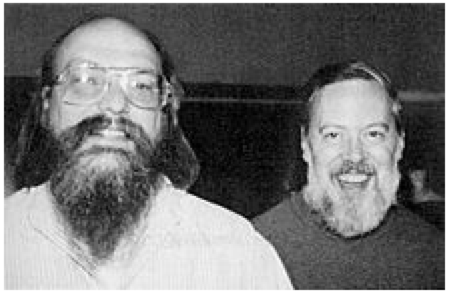


The Go mascot

Some Go highlights

- Produces machine code, can call C code
- Garbage collection
- Dynamic typing
- Handy data structures in basic language:
  - Strings, arrays, slices, maps
  - Many more in libraries
- Pointers but no pointer arithmetics
- Object oriented but classes optional
- Easy concurrency
- Comes with build tools

Rob Pike



Ken Thomson (L)
with
Dennis Ritchie

Backwards compatibility

- Breaking existing code is a serious crime in the Go world
- For maintaining old code, go ships the `fix` tool that:
  - Looks for deprecated language constructs in Go itself
  - Looks for old API calls in library calls
  - Fixes the code to use modern syntax/API.
- The tool needs config info from libraries. This is part of shipping a new version of a Go library.

More useful tools

- `gofmt` formats Go source code the One True Way. No more coding style fights!
- `godoc` extracts docs, creates HTML.
- `vet` statically analyzes source code for suspicious signs.

A taste of Go

- This talk is meant to give you a quick overview of what go looks like.
- We won't discuss advanced concepts such as OOP
- You can code in Go without OOP but you should use OO for large code
- If you know C or Java, this will look very familiar.

Shut up and show me some code

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, world!")
}
```

- Hey, look, no semicolon. But brackets, so yay.
- All Go code is organized in packages.
- A program entry point is function main() in package main.

Export conventions

```
package main

import (
    "fmt"
    "math"
)

func main() {
    fmt.Printf("Have some pi: %g", math.Pi)
}
```

- Variables starting with a capital are exported automatically.
- All others are internal.
- Here, package `math` exports constant `Pi`.
- Look, Ma, no need for complex declarations!

Functions, strings, array

- Function args and return type is defined in definition
- Type comes after the variable name(s)
- Variables have dynamic types. Here, `s` is an array of strings
- The `:=` construct is used to create a new variable on the fly.

```
$ cat prog3.go
package main

import (
    "fmt"
    "strings"
)

func compliment(x, y, z string) string {
    s := []string{x, y, z}
    return strings.Join(s, " is a ")
}

func main() {
    fmt.Println(compliment("A rose",
"rose", "rose"))
}
$ go run prog3.go
A rose is a rose is a rose
```

Variables, auto type, multiple return values

- You can declare variables with `var` + name and type
- Or you can use `:=` on a new var and let the compiler infer the type.
- Functions can return multiple values
- Lists are simple!
- Undefined vars are initialized to 0 (int), false (bool), "" (string)

```
$ cat prog4.go
package main

import "fmt"

func values_and_sum(x, y int) (int, int,
int) {
	return x, y, x + y
}

func main() {
	var a, b int
	a = 3
	b = 2
	c := a + b
	fmt.Printf("%d + %d = %d\n",
	    a, b, c)
	x, y, z := values_and_sum(4, 6)
	fmt.Printf("%d + %d = %d\n",
	    x, y, z)
}
$ go run prog4.go
3 + 2 = 5
4 + 6 = 10
```

Loops

- While and do... until are for sissies.
- There is only for. All hail for!
- "for condition" is the same as "while condition"
- "for {... }" is the same as "while true {...}"

```
$ cat prog6.go
package main

import "fmt"

func main() {
    sum := 0
    for i := 0; i < 10; i++ {
        sum += i
    }
    fmt.Println(sum)
    sum = 0
    for sum < 100 {
        sum++
    }
    fmt.Println(sum)
}
$ go run prog6.go
45
100
```

Range loops

- The range operator returns a current index and current value of an array or map

```
$ cat prog8.go
package main

import "fmt"

var pow = []int{1, 2, 4, 8, 16, 32}

func main() {
    for i, v := range pow {
        fmt.Printf("2**%d = %d\n", i, v)
    }
}
$ go run prog8.go
2**0 = 1
2**1 = 2
2**2 = 4
2**3 = 8
2**4 = 16
2**5 = 32
```

Other constructs

- You also have if…else and switch…case

- The "defer *statement*" executes *statement* when the function exits. Equivalent of "on exit".

- Structs like in C

- Maps

```
$ cat prog9.go
package main

import "fmt"

type Person struct {
    name string
    age int
}

func main() {
    joe := Person{"Joe", 25}
    fmt.Println(joe)
    job := make(map[string]Person)
    job["Sysadmin"] = Person{"Fred", 39}
    fmt.Println(job["Sysadmin"])
}

$ go run prog7.go
{Joe 25}
{Fred 39}
```

Questions?