

The GO Programming Language

Chris Lupo

Presented to:

CPLUG

Cal Poly, San Luis Obispo

February 9, 2010

Outline

- Introduction
- GO Basics
- Tools
- Project Ideas

Introduction

Go is:

- New
 - Started by Robert Griesemer, Ken Thompson, and Rob Pike (Googlers) in late 2007.
 - Released in October of 2009.
 - Still very young.

Introduction

Go is:

- Concurrent
 - Go promotes writing systems and servers as sets of lightweight communicating processes, called *goroutines*. Run thousands of goroutines if you want—and say good-bye to stack overflows.
 - Language takes care of goroutine management, memory management.
 - Growing stacks, multiplexing of goroutines onto threads is done automatically.

Introduction

Go is:

- Garbage-collected
 - Concurrency is hard without garbage-collection
 - Garbage-collection is hard without the right language
 - GO's implementation is efficient and latency-free

Introduction

Go is:

- A systems programming language
 - First in over a decade
 - In that decade, we've had:
 - sprawling libraries & dependency chains
 - dominance of networking
 - client/server focus
 - massive clusters
 - the rise of multi-core CPUs
 - C/C++ were not designed with these in mind

Introduction

Go is:

- Fast
 - It takes too long to build software.
 - The tools are slow and are getting slower.
 - Dependencies are uncontrolled.
 - Machines have stopped getting faster.
 - Yet software still grows and grows.

Introduction

Go is:

- Fun?! Easy?!
 - The language is simple, so it's easier to be productive in
 - Few keywords, parsable without symbol table.
 - No stuttering; don't want to see

```
foo.Foo *myFoo = new foo.Foo(foo.FOO_INIT)
```

- Keep the type system clear. No type hierarchy. Too clumsy to write code by constructing type hierarchies.
- It can still be object-oriented.

Hello, world

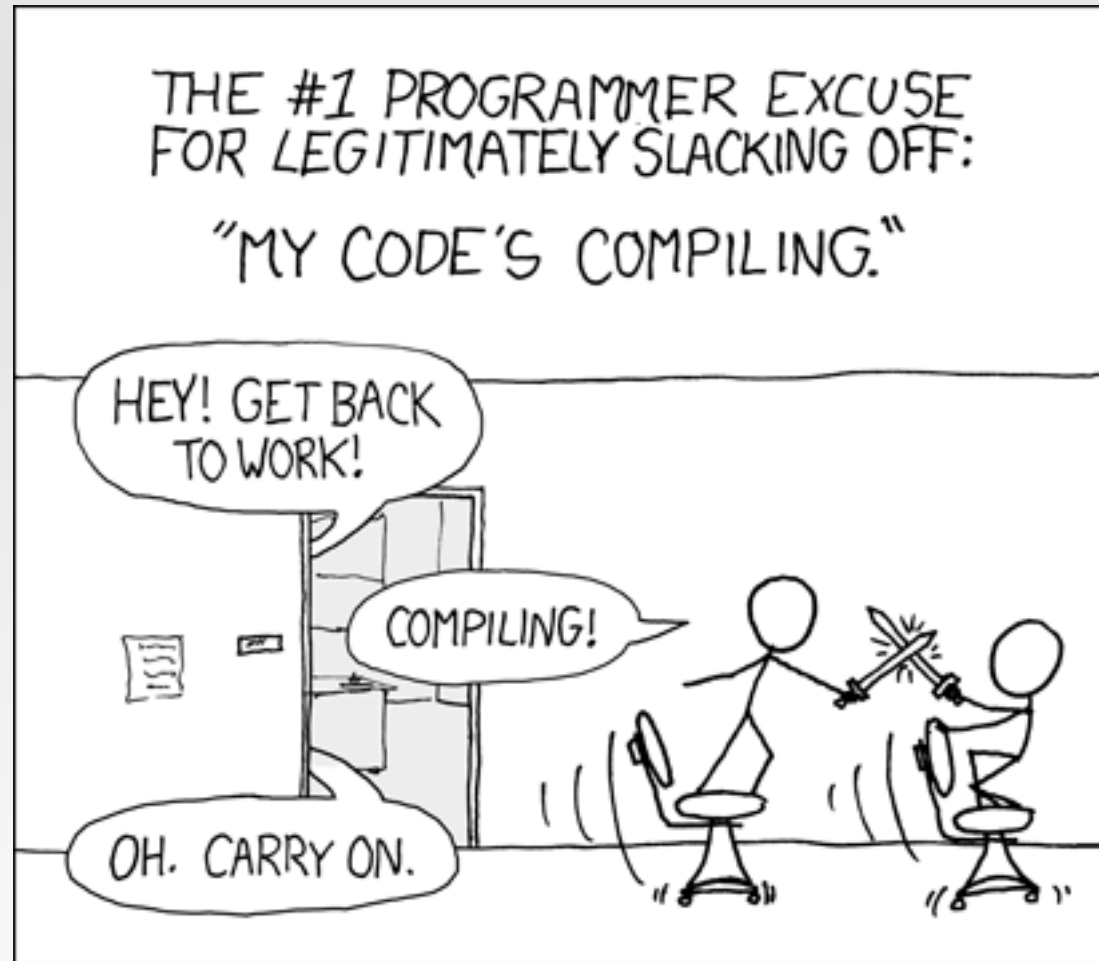
```
package main

import "fmt"

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

Go Fast!

As xkcd observes:



<http://xkcd.com/303/>

GO Basics

- Comments
 - Both C-style `/* */` and C++-style `//` comments supported
 - `//` are the norm, except for block comments in packages and for disabling large chunks of code

GO Basics

- Formatting
 - Indentation
 - Use tabs for indentation. **gofmt** will handle alignment for you.
 - Parentheses
 - Go needs fewer parentheses: control structures (if, for, switch) don't require parentheses. Also, the operator precedence hierarchy is shorter and clearer, so
$$x \ll 8 + y \ll 16$$
means what the spacing implies.

GO Basics

- Semicolons
 - Don't need 'em! (usually)
 - Typically only seen separating the clauses of for loops and the like
 - makes for clean-looking, semicolon-free code.
 - The one surprise is that it's important to put the opening brace of a construct such as an `if` statement on the same line as the `if`; if you don't, there are situations that may not compile or may give the wrong result.

GO Basics

- `if` statements

- In Go a simple if looks like this:

```
if x > 0 {  
    return y  
}
```

- `if` accepts an initialization statement, used to set up a local variable.

```
if err := file.Chmod(0664); err != nil {  
    log.Stderr(err)  
    return err  
}
```

GO Basics

- Loops
 - One structure that unifies `for` and `while`, there is no `do-while`. Comes in three formats:
 - // Like a C `for`
`for init; condition; post { }`
 - // Like a C `while`
`for condition { }`
 - // Like a C `for(;;)`
`for { }`

GO Basics

- Loops
 - Go has no comma operator and ++ and -- are statements not expressions, if you want to run multiple variables in a for you should use parallel assignment.

```
// Reverse a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
    a[i], a[j] = a[j], a[i]
}
```

GO Basics

- Switch
 - Go's `switch` is more general than C's.
 - The expressions need not be constants or even integers, the cases are evaluated top to bottom until a match is found, and if the `switch` has no expression it switches on `true`.
 - It's therefore possible—and idiomatic—to write an `if-else-if-else` chain as a `switch`.

Go Basics

- Switch

```
func unhex(c byte) byte {  
    switch {  
    case '0' <= c && c <= '9':  
        return c - '0'  
    case 'a' <= c && c <= 'f':  
        return c - 'a' + 10  
    case 'A' <= c && c <= 'F':  
        return c - 'A' + 10  
    }  
    return 0  
}
```

Go Basics

- Switch

There is no automatic fall through, but cases can be presented in comma-separated lists.

```
func shouldEscape(c byte) bool {  
    switch c {  
        case ' ', '?', '&', '=', '#', '+', '%':  
            return true  
        }  
        return false  
    }  
}
```

Go Basics

- Types
 - Go has some familiar types such as `int` and `float`, which represent values of the "appropriate" size for the machine.
 - It also defines explicitly-sized types such as `int8`, `float64`, and so on, plus unsigned integer types such as `uint`, `uint32`, etc. These are distinct types; even if `int` and `int32` are both 32 bits in size, they are not the same type.
 - There is also a byte synonym for `uint8`, which is the element type for strings.

Go Basics

- Strings

- Yes! That's a built-in. Strings are *immutable values*, not just arrays of byte values. Once you've built a string value, you can't change it.
- You can change a string variable simply by reassigning it. This snippet is legal code:

```
s := "hello"
if s[1] != 'e' { os.Exit(1) }
s = "good bye"
var p *string = &s
*p = "ciao"
```

Go Basics

- Arrays

- Arrays are declared like this:

```
var arrayOfInt [10]int;
```

- Arrays, like strings, are values, but they are mutable.
- This differs from C, in which `arrayOfInt` would be usable as a pointer to `int`.
- In Go, since arrays are values, it's meaningful (and useful) to talk about pointers to arrays.

Go Basics

- Pointers
 - GO has them, but they're limited
 - No pointer arithmetic
 - Easier for garbage-collection

Advanced GO

- Functions
- Methods
- Structs
- Goroutines
- Channels
- Etc.
 - Read the Tutorial for more examples and details

Tools

- Two compilers
 - 6g/8g/5g (Ken Thompson)
 - more experimental.
 - generates OK code very quickly.
 - not GCC-linkable but has FFI support.
 - gccgo (Ian Taylor)
 - Go front end for GCC.
 - generates good code not as quickly.
- Both support 32- and 64-bit x86, plus ARM.
- Performance: typically within 10-20% of C.

Tools

- Run-time
 - Run-time handles memory allocation and collection, stack handling, goroutines, channels, slices, maps, reflection, and more.
 - Solid but improving.
 - 6g has good goroutine support, muxes them onto threads, implements segmented stacks.
 - Gccgo is (for a little while yet) lacking segmented stacks, allocates one goroutine per thread.

Tools

- Garbage-collector
 - 6g has a simple but effective mark-and-sweep collector. Work is underway to develop the ideas in IBM's Recycler garbage collector* to build a very efficient, low-latency concurrent collector.
 - Gccgo at the moment has no collector; the new collector is being developed for both compilers.

Tools

- Libraries

Lots of libraries but plenty still needed.

- Some (e.g. regexp) work fine but are too simple.
- OS, I/O, files
- math (sin(x) etc.)
- strings, Unicode, regular expressions
- reflection
- command-line flags, logging
- hashes, crypto
- testing (plus testing tool, gotest)
- networking, HTTP, RPC
- HTML (and more general) templates

Tools

- Godoc and Gofmt

- Godoc:

documentation server, analogous to javadoc but easier on the programmer. Can run yourself but live at:

- <http://golang.org/> (top-level; serves all docs)
- <http://golang.org/pkg/> (package docs)
- <http://golang.org/src/> (source code)

- Gofmt:

pretty-printer; all code in the repository has been formatted by it.

Tools

- Debugger
- A custom debugger is underway; not quite ready yet (but close).
- Gccgo users can invoke gdb but the symbol table makes it look like C and there's no knowledge of the run-time.

Conclusions

- It's early yet but promising.
- A very comfortable and productive language.
- Lots of documents on the web: specification, tutorial, "Effective Go", FAQs, more
- Full open-source implementations.
 - Want to try it?
 - Want to help?
 - Want to build libraries or tools?

<http://golang.org>

Project Ideas

- Did you know we have a cluster?
 - And that it's under-utilized?
- I'll advise GO related projects that explore concurrency, performance, code size, etc.
 - Especially interested in ARM, but do what makes sense...