Introduction to Go

Kevin W. Gisi

Twin Cities Code Camp 8—April 10, 2010

Summary

Go is a brand-spanking-new systems language that Google released in November, 2009. Every wonder how awesome C would be if it was garbage-collected, concurrent, and didn't take a few weeks to compile? Wake up; it's here! We'll take a look at this new language that steals some of the dynamic flexibility of Python and Ruby, the performance of C, and a compile time that you'll miss if you blink.

Hello, world

```
package main

import "fmt"

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

code/hello_world.go

Why Go?

- It's a systems language
- It's fun, like dynamic languages

We Already Have a Systems Language!

Like C

```
void        (int m, int t, int
c) {
  ((t / m) <= 1) ?
primes(m,t+1,c) : !(t % m) ?
primes(m,t+1, t % m) :
  ((t % m)==(t / m) && !c) ?
(printf("%d\t",(t / m)),
primes(m,t+1,c)) :
  ((t % m)> 1 && (t % m) < (t /
m)) ? primes(m,t+1,c + !((t /
m) % (t % m))) :
  (t < m * m) ? primes(m,t+1,c)
: 0;
}
```

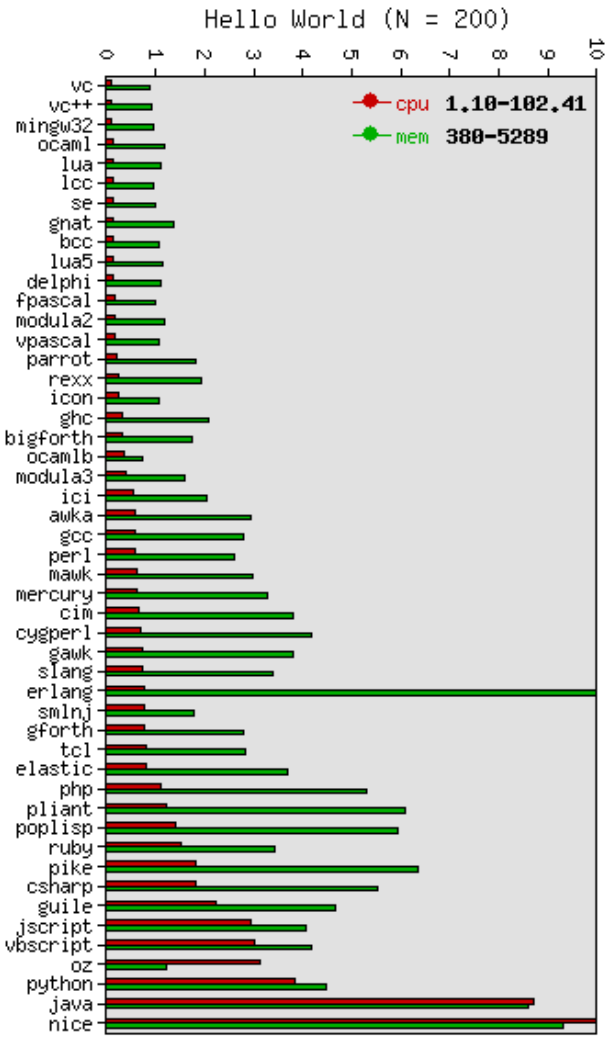
code/c.c

We Already Have Fun Languages!

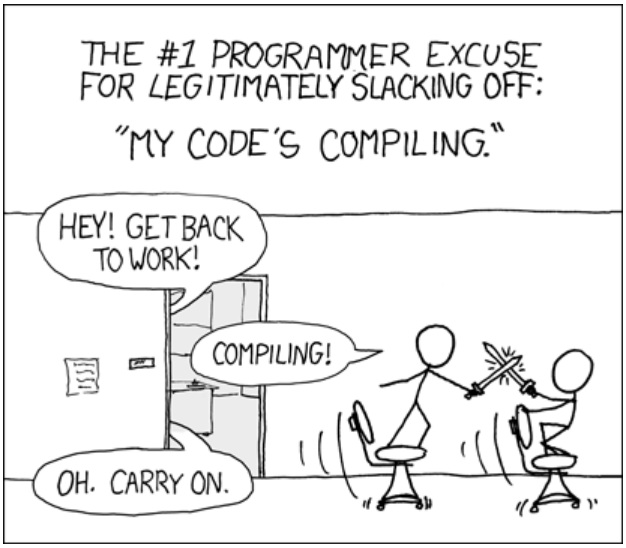
```
module
  def                ( *  )
    args.       ? cmd.     :
"#{cmd} #{args.    (" ")}"
  end
end

puts this is terrible code

x = [1,2,3]
    (x.    ('+'))
```

code/ruby.rb

Fun Languages are Slow at Runtime



Fast Languages are Slow to Compile



Alternative: Go

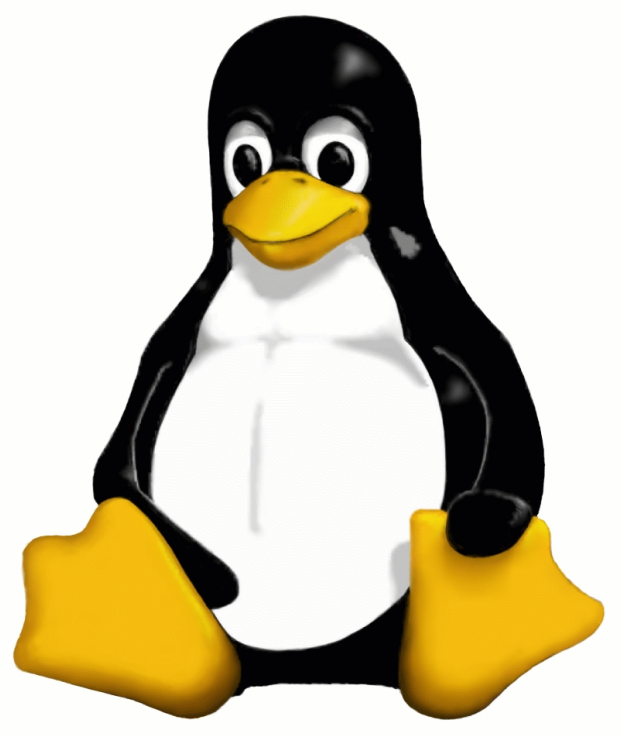
~~Go is slow at runtime AND at compile time!~~

- 1/2—1/3 LOC for C++ benchmarks
- 1—1/16 of the speed (libraries?)

Priorities:

- Compile time
- Performance
- Readability

It Runs on Linux



It Runs on OS X!

It Runs on Virtual Machines!

Specifications

- Compiled
- Imperative, structured
- Concurrent
- Strongly typed (explicit or inferred)

Variables & Types

- int, float
- int8, int32, float64
- uint, ufloat
- string
- struct

Variables: Pointers and Arrays

Pointers

- Use them for reference
- DON'T manipulate them!

Arrays

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

code/variables.go

Variables: Slices and Maps

Slices

- "Pointers" for arrays
- Contains pointers to each object in a range of the array
- Used for passing array values by reference

Maps

```
var m map = map[string] int{}
m["price"] = 5
```

code/variables.go

Variable Declaration

```
// Declare a variable
var s string = "";

// Go infers the type
var s2 = "";

// Syntactic shorthand - initializing declaration
s3 := "";
```

code/variables.go

Variable Allocation

new()

- allocates heap space
- zero-initializes the space
- returns the address

make()

- allocates heap space
- creates the object (and underlying data structure)
- returns the value

Go ≠ C

- Semicolons optional (implied)
- Curly braces MUST start on the same line
- No parentheses in `ifs` and `fors`
- Garbage collected
- Arrays aren't pointers

Functions

- CamelCase - public
- camelCase - package-level
- Pass by value
- Multiple return values

What? Multiple Return Values?!

```
func           () (int, string) {
  return 5, "thanks for asking!"
}
```

code/functions.go

How do we access them?

```
x, message := gimmeFive()
```

code/functions.go

Why Multiple Return Values?

```
  if text, err :=
readFile('foo.txt'); err == nil
{
    // Read the file
  } else {
    // Handle the error
  }

  for key, value := range
my_map {

  }

  // ...or
  for _, value := range my_map
{ // Discard the first returned
value

  }
```

code/functions.go

Named Results

```
func          () (value int, err
string) {
  // value and err are set to
nil
  value = 5
  return // implicitly returns
5, nil
}
```

code/functions.go

Methods

Write functions that operate on particular data

```
  type Mongoose int

  func (           ) Pluck()
string {
    return "The mongoose was
plucked"
  }
```

code/functions.go

Interfaces

- Similar to Java
- Definition of acceptable types
- Types don't "implement" anything

Interfaces

```
type Duck interface {
  Quack()
  Waddle()
  Swim()
}

func (i int)       () {}
func (i int)        () {}
func (i int)      () {}

func (d Duck)           () {} //
We can pass this an int
```

code/interfaces.go

Interface Embedding

```
type Spider interface {
  CreateWebs()
  ClimbWalls()
}

type Man interface {
  TakePhotos()
  MissUncleBen()
}

type SpiderMan interface {
  Spider
  Man
}
```

code/interfaces.go

Struct Embedding

```
struct Spider interface {
  CreateWebs()
  ClimbWalls()
}

struct Man interface {
  TakePhotos()
  MissUncleBen()
}

struct SpiderMan interface {
  Spider
  Man
}

func (m Man)          () {} //
Can pass a Man or a SpiderMan
```

code/interfaces.go

Concurrency

Goroutines

- NOT threads
- Independent code
- Communication over shared memory

Threads

- Exist
- Span across multiple cores
- Go load-balances them
- Don't worry about it

Goroutine Example

```
  go gimmeFive()

  go func() {
    time.Sleep(5)
    fmt.Println("Computer over.
Virus = very yes")
  }()
```

code/concurrency.go

- No access to spawned goroutines
- No thread.join equivalent

Channels

- Like Unix pipes
- Communicate across goroutines
- Optionally blocking/non-blocking

Channel Example: Communication

```
func      () {
  output := make(chan string)
  go log(output)
  for true {
    output <- "Hey. Hey. Listen"
  }
}

func     (                  ){
  for true {
    data := <- input
    fmt.Println(data)
  }
}
```

code/concurrency.go

Channel Example: Asynchronous

```
func         (){
  output := make(chan string,
10)
  go log(output)
  output <- "Hey. Hey. Listen"
  // Chatter terminates almost
immediately
}
```

code/concurrency.go

Channel Example: Semaphore

```
func        (               ) {
  for true {
    <- input
    // Do something
    input <- 1
  }
}
```

code/concurrency.go

Channel Example: Thread.join

```
func         (){
  gui_dead := make(chan int)
  go gui()

  // Do some stuff
  <- gui_dead
}
```

code/concurrency.go