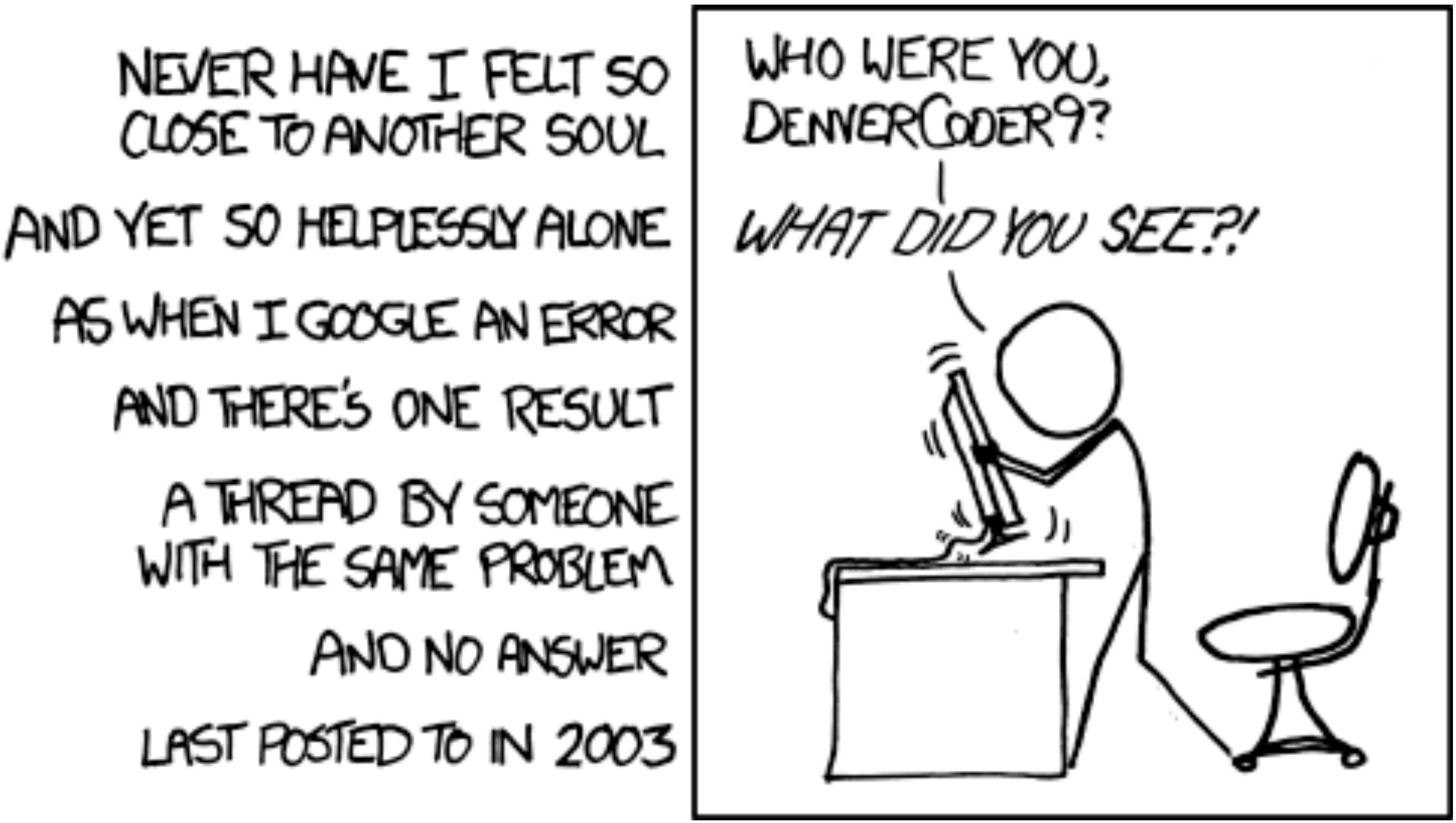
Lecture 19: Go



http://xkcd.com/979/

Go

- developed ~2007 at Google by
  Robert Griesemer, Rob Pike, Ken Thompson
- open sourced in 2009

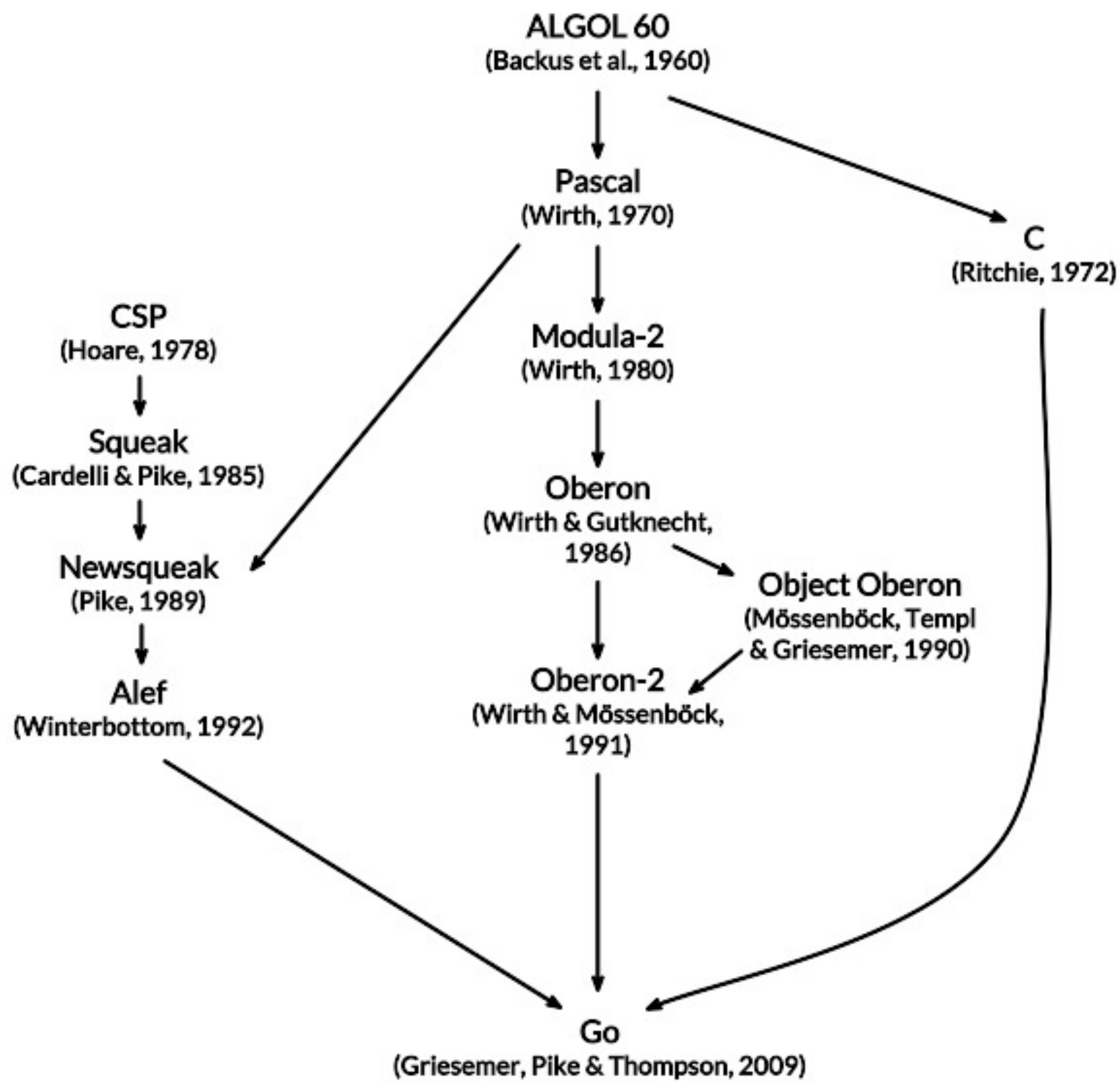
- compiled, statically typed
  - very fast compilation
- C-like syntax
- garbage collection
- built-in concurrency

- no classes or type inheritance or overloading or generics
  - unusual interface mechanism instead of inheritance

Outline

- history
- basic constructs, simple programs
- arrays & slices
- maps
- methods, interfaces
- concurrency, goroutines

Influences


ALGOL 60
(Backus et al., 1960)
Pascal
(Wirth, 1970)
C
(Ritchie, 1972)
CSP
(Hoare, 1978)
Modula-2
(Wirth, 1980)
Squeak
(Cardelli & Pike, 1985)
Oberon
(Wirth & Gutknecht, 1986)
Newsqueak
(Pike, 1989)
Object Oberon
(Mössenböck, Templ & Griesemer, 1990)
Alef
(Winterbottom, 1992)
Oberon-2
(Wirth & Mössenböck, 1991)
Go
(Griesemer, Pike & Thompson, 2009)

Hello world in Go

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, 世界")
}
```

```
$ go run hello.go     # to compile and run
$ go build hello.go   # to create a binary

$ go help             # for more
```

Types, constants, variables

- basic types

  `bool string  int8 int16 int32 int64  uint8 …  int  uint`

  `float32 float64 complex64 complex128`

  quotes: `'世'`, `"UTF-8 string"`, `` `raw string` ``

- variables and declarations

```
var c1, c2 rune
var x, y, z = 0, 1.23, false   // variable decls

x := 0; y := 1.23; z := false  // short variable decl
```

  Go infers the type from the type of the initializer

  assignment between items of different type requires an explicit conversion, e.g., int(float_expression)

- operators
  - mostly like C, but `++` and `--` are postfix only and are not expressions
  - assignment is not an expression
  - no `?:` operator

Echo command:

```
// Echo prints its command-line arguments.
package main

import (
  "fmt"
  "os"
)

func main() {
    var s, sep string
    for i := 1; i < len(os.Args); i++ {
        s += sep + os.Args[i]
        sep = " "
    }
    fmt.Println(s)
}
```

Echo command (version 2):

```
// Echo prints its command-line arguments.
package main

import (
  "fmt"
  "os"
)

func main() {
    s, sep := "", ""
    for _, arg := range os.Args[1:] {
        s += sep + arg
        sep = " "
    }
    fmt.Println(s)
}
```

Statements, control flow: `if-else`

- statements
  - assignment, control flow, function call, …
  - scope indicated by mandatory braces;   no semicolon terminator needed

- control flow: if-else, for, switch, ...

```
if opt-stmt; boolean {
    statements
} else if opt-stmt; boolean {
    statements
} else {
    statements
}
```

```
if c := f.ReadByte(); c != EOF { // scope of c is if-else
    ...
}
```

Control flow: `switch`

```
switch opt-stmt; opt-expr {
case exprlist: statements    // no fallthrough
case exprlist: statements
default: statements
}
```

```
switch Suffix(file) {
case ".gz":  return GzipList(file)
case ".tar": return TarList(file)
case ".zip": return ZipList(file)
}
```

```
switch {
case Suffix(file) == ".gz": ...
}
```

– can also switch on types

Control flow: `for`

```
for opt-stmt; boolean; opt-stmt {
    statements  // break, continue   (with optional labels)
}

for boolean {
    // while statement
}

for {
    // infinite loop
}

for index, value := range something {
    // ...
}
```

Arrays and slices

- an array is a fixed-length sequence of same-type items

```
months := [...]string {1:"Jan", 2:"Feb", /*...,*/ 12:"Dec"}
```

- a slice is a subsequence of an array

```
summer := months[6:9]; Q2 := months[4:7]
```

- elements accessed as `slice[index]`
  - indices from 0 to `len(slice)-1` inclusive

`summer[0:3]` is elements `months[6:9]`

```
summer[0] = "Juin"
```

- loop over a slice with for range

```
for i, v := range summer {
   fmt.Println(i, v)
}
```

- slices are very efficient (represented as small structures)
- most library functions work on slices

Maps (== associative arrays)

- unordered collection of key-value pairs
  - keys are any type that supports `==` and `!=` operators
  - values are any type

```
// Find duplicated lines in stdin.
func main() {
  counts := make(map[string]int)
  in := bufio.NewScanner(os.Stdin)
  for in.Scan() {
     counts[in.Text()]++
  }
  for line, n := range counts {
     if n > 1 {
          fmt.Printf("%d\t%s\n", n, line)
     }
  }
}
```

Methods and pointers

- can define methods that work on any type, including your own:

```
type Vertex struct {
    X, Y float64
}
func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}
func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
func main() {
    v := &Vertex{3, 4}
    v.Scale(5)
    fmt.Println(v, v.Abs())
}
```

Interfaces

- an interface is satisfied by any type that implements all the methods of the interface
- completely abstract: can't instantiate one
- can have a variable with an interface type
- then assign to it a value of any type that has the methods the interface requires

- a type implements an interface merely by defining the required methods
  - it doesn't declare that it implements them

- Writer: the most common interface

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Example of Writer interface

```
type ByteCounter int

func (c *ByteCounter) Write(p []byte) (int, error) {
  *c += ByteCounter(len(p)) // convert int to ByteCounter
  return len(p), nil
}
func main() {
  var c ByteCounter
  c.Write([]byte("hello"))
  fmt.Println(c) // "5", = len("hello")

  c = 0 // reset the counter

  var name = "Bob"
  fmt.Fprintf(&c, "hello, %s", name)
  fmt.Println(c) // "10", = len("hello, Bob")
}
```

Sort interface

- sort interface defines three methods
- any type that implements those three methods can sort
- algorithms are inside the sort package, invisible outside

```
package sort

type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

```
// Len and Swap already defined for slices:
sort.Slice(sub, func(i, j int) bool {
    return sub[i].Subname < sub[j].Subname
})
```

Sort interface (adapted from Go Tour)

```
type Person struct {
  Name string
  Age  int
}
func (p Person) String() string {
  return fmt.Sprintf("%s: %d", p.Name, p.Age)
}
type ByAge []Person

func (a ByAge) Len() int           { return len(a) }
func (a ByAge) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }
func (a ByAge) Less(i, j int) bool { return a[i].Age < a[j].Age }

func main() {
  people := []Person{{"Bob",31}, {"Sue",42}, {"Ed",17}, {"Jen",26},}
  fmt.Println(people)
  sort.Sort(ByAge(people))
  fmt.Println(people)
}
```

Tiny web server

```
func main() {
  http.HandleFunc("/", handler)
  http.ListenAndServe("localhost:8000", nil)
}

// handler echoes Path component of the request URL r.
func handler(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}
```

`http.ResponseWriter` implements Writer interface

Tiny version of curl

```
func main() {
  url := os.Args[1]
  resp, err := http.Get(url)
  if err != nil {
      fmt.Fprintf(os.Stderr, "curl: %v\n", err)
      os.Exit(1)
  }
  _, err = io.Copy(os.Stdout, resp.Body)
  if err != nil {
      fmt.Fprintf(os.Stderr, "curl: copying %s: %v\n",
                                       url, err)
      os.Exit(1)
  }
}
```

Concurrency: goroutines & channels

- **channel: a type-safe generalization of Unix pipes**
  - inspired by Hoare's Communicating Sequential Processes (1978)
- **goroutine: a function executing concurrently with other goroutines in the same address space**
  - run multiple parallel computations simultaneously
  - loosely like threads but much lighter weight
- **channels coordinate computations by explicit communication**
  - locks, semaphores, mutexes, etc., are less often used

Example: web crawler

- **want to crawl a bunch of web pages to do something**
  – e.g., figure out how big they are
- **problem: network communication takes relatively long time**
  – program does nothing useful while waiting for a response
- **solution: access pages in parallel**
  – send requests asynchronously
  – display results as they arrive
  – needs some kind of threading or other parallel process mechanism
- **takes less time than doing them sequentially**

Version 1: no parallelism

```
func main() {
  start := time.Now()
  for _, site := range os.Args[1:] {
    count("http://" + site)
  }
  fmt.Printf("%.2fs total\n", time.Since(start).Seconds())
}

func count(url string) {
  start := time.Now()
  r, err := http.Get(url)
  if err != nil {
    fmt.Printf("%s: %s\n", url, err)
    return
  }
  n, _ := io.Copy(ioutil.Discard, r.Body)
  r.Body.Close()
  dt := time.Since(start).Seconds()
  fmt.Printf("%s %d [%.2fs]\n", url, n, dt)
}
```

Version 2: parallelism with goroutines

```
func main() {
  start := time.Now()
  c := make(chan string)
  n := 0
  for _, site := range os.Args[1:] {
    n++
    go count("http://" + site, c)
  }
  for i := 0; i < n; i++ {
    fmt.Print(<-c)
  }
  fmt.Printf("%.2fs total\n", time.Since(start).Seconds())
}
func count(url string, c chan<- string) {
  start := time.Now()
  r, err := http.Get(url)
  if err != nil {
    c <- fmt.Sprintf("%s: %s\n", url, err)
    return
  }
  n, _ := io.Copy(ioutil.Discard, r.Body)
  r.Body.Close()
  dt := time.Since(start).Seconds()
  c <- fmt.Sprintf("%s %d [%.2fs]\n", url, n, dt)
}
```

Python version, no parallelism

```
import urllib2, time, sys

def main():
  start = time.time()
  for url in sys.argv[1:]:
    count("http://" + url)
  dt = time.time() - start
  print "\ntotal: %.2fs" % (dt)

def count(url):
  start = time.time()
  n = len(urllib2.urlopen(url).read())
  dt = time.time() - start
  print "%6d  %6.2fs   %s" % (n, dt, url)

main()
```

Python version, with threads

```
import urllib2, time, sys, threading

global_lock = threading.Lock()

class Counter(threading.Thread):
  def __init__(self, url):
    super(Counter, self).__init__()
    self.url = url

  def count(self, url):
    start = time.time()
    n = len(urllib2.urlopen(url).read())
    dt = time.time() - start
    with global_lock:
      print "%6d  %6.2fs   %s" % (n, dt, url)

  def run(self):
    self.count(self.url)

def main():
  threads = []
  start = time.time()
  for url in sys.argv[1:]:  # one thread each
    w = Counter("http://" + url)
    threads.append(w)
    w.start()

  for w in threads:
    w.join()
  dt = time.time() - start
  print "\ntotal: %.2fs" % (dt)

main()
```

Where will Go go?

- comparatively small but rich language

- efficient; compilation is very fast
- concurrency model is convenient and efficient
- object model is unusual but seems powerful

- significant use at Google and elsewhere
- most often for web server applications

- "C for the 21st century" ?

Go source materials

- official web site:
  golang.org

- Go tutorial, playground

- Rob Pike on why it is the way it is:
  http://www.youtube.com/watch?v=rKnDgT73v8s

- Russ Cox on interfaces, reflection, concurrency
  http://research.swtch.com/gotour