The
Go
Programming Language

Part 1

Rob Pike
r@google.com
(updated June 2011)

Who

Work done by a small team at Google, plus lots of contributors from around the world.

Contact:

http://golang.org:	web site
golang-nuts@golang.org:	user discussion
golang-dev@golang.org:	developers

Course Outline

Day 1
  Basics

Day 2
  Types, methods, and interfaces

Day 3
  Concurrency and communication

This course is about programming in Go, not about programming language design. That is the topic of a separate talk, not yet on line.

Google

Today's Outline

Motivation

Basics
  the easy, mostly familiar stuff

Packages and program construction

Google

Motivation

Google

Why a new language?

In the current world, languages don't help enough:

Computers fast but software construction slow.

Dependency analysis necessary for speed, safety.

Types get in the way too much.

Garbage collection, concurrency poorly supported.

Multi-core seen as crisis not opportunity.

Google

To put it in a positive way

Our goal is to make programming fun again.

- the feel of a dynamic language with the safety of a static type system

- compile to machine language so it runs fast

- real run-time that supports GC, concurrency

- lightweight, flexible type system

- has methods but not a conventional OO language

Google

Resources

For more background, etc. see the documentation:

http://golang.org/

Includes:

- language specification
- tutorial
- "Effective Go"
- library documentation
- setup and how-to docs
- FAQs
- a playground (run Go from the browser)
- more

Google

Status: Compilers

gc (Ken Thompson), a.k.a. 6g, 8g, 5g
- derived from the Plan 9 compiler model
- generates OK code very quickly
- not directly gcc-linkable

gccgo (Ian Taylor)
- more familiar architecture
- generates good code not as quickly
- gcc-linkable

Available for 32-bit and 64-bit x86 (amd64, x86-64) and ARM.

Garbage collector, concurrency etc. implemented.
Good and improving libraries.

Google

Basics

Google

Time for some code

```
package main

import "fmt"

func main() {
    fmt.Print("Hello, 世界\n")
}
```

Google

Language basics

Assuming familiarity with other C-like languages, here comes a quick tour of the basics.

This will be mostly easy, familiar and therefore dull. Apologies for that.

The next two lectures contain the fun stuff but we need to lay down the groundwork first.

Google

Lexical structure

Mostly traditional with modern details.

Source is UTF-8. White space: blank, tab, newline, carriage return.

Identifiers are letters and numbers (plus '_') with "letter" and "number" defined by Unicode.

Comments:

```
/* This is a comment; no nesting */
// So is this.
```

Literals

C-like but numbers require no signedness or size markings (more about this soon):

```
23
0x0FF
1.234e7
```

C-like strings, but Unicode/UTF-8. Also `\xNN` always 2 digits; `\012` always 3; both are bytes:

```
"Hello, world\n"
"\xFF"    // 1 byte
"\u00FF"  // 1 Unicode char, 2 bytes of UTF-8
```

Raw strings:

```
`\n\.abc\t\` == "\\n\\.abc\\t\\"
```

Syntax overview

Basically C-like with reversed types and declarations, plus keywords to introduce each type of declaration.

```
var a int
var b, c *int  // note difference from C
var d []int
type S struct { a, b int }
```

Basic control structures are familiar:

```
if a == b { return true } else { return false }
for i = 0; i < 10; i++ { ... }
```

Note: no parentheses, but braces required.

More about all this later.

Google

Semicolons

Semicolons terminate statements but:

- lexer inserts them automatically at end of line if the previous token could end a statement.
- Note: much cleaner, simpler than JavaScript rule!

Thus no semis needed in this program:

```
package main

const three = 3
var i int = three

func main() { fmt.Printf("%d\n", i) }
```

In practice, Go code almost never has semicolons outside `for` and `if` clauses.

Google

Numeric types

Numeric types are built in, will be familiar:

`int`	`uint`		
`int8`	`uint8 = byte`		
`int16`	`uint16`		
`int32`	`uint32`	`float32`	`complex64`
`int64`	`uint64`	`float64`	`complex128`

Also `uintptr`, an integer big enough to store a pointer.

These are all distinct types; `int` is not `int32` even on a 32-bit machine.

No implicit conversions (but don't panic).

Bool

The usual boolean type, `bool`, with values `true` and `false` (predefined constants).

The `if` statement etc. use boolean expressions.

Pointers and integers are not booleans.†

† Consider (not Go): `const bool False = "false";`

String

The built-in type `string` represents immutable arrays of bytes – that is, text. Strings are length-delimited not NUL-terminated.

String literals have type `string`.

Immutable, just like `int`s. Can reassign variables but not edit values.

Just as 3 is always 3, `"hello"` is always `"hello"`.

Language has good support for string manipulation.

Google

Expressions

Mostly C-like operators.
Binary operators:

Prec.	operators	comments
5	* / % << >> & &^	&^ is "bit clear"
4	+ - | ^	^ is "xor"
3	== != < <= > >=	
2	&&	
1	||	

Operators that are also unary: & ! * + - ^ (plus <- for communication).
Unary ^ is complement.

Google

Go vs. C expressions

Surprises for the C programmer:

fewer precedence levels (should be easy)
^ instead of ~ (it's binary "exclusive or" made unary)
++ and -- are not expression operators
(x++ is a statement, not an expression;
*p++ is (*p)++ not *(p++))
&^ is new; handy in constant expressions
<< >> etc. require an unsigned shift count

Non-surprises:
assignment ops work as expected: += <<= &^= etc.
expressions generally look the same (indexing, function call, etc.)

Examples

```
+x
23 + 3*x[i]
x <= f()
^a >> b
f() || g()
x == y + 1 && <-ch > 0
x &^ 7  // x with the low 3 bits cleared
fmt.Printf("%5.2g\n", 2*math.Sin(PI/8))
7.234/x + 2.3i

"hello, " + "world"  // concatenation
                     // no C-like "a" "b"
```

Numeric conversions

Converting a numeric value from one type to another is a conversion, with syntax like a function call:

```
uint8(intVar)      // truncate to size
int(float64Var)    // truncate fraction
float64(intVar)    // convert to float64
```

Also some conversions to and from `string`:

```
string(0x1234)             // == "\u1234"
string(sliceOfBytes)       // bytes -> bytes
string(sliceOfInts)        // ints -> Unicode/UTF-8
[]byte("abc")              // bytes -> bytes
[]int("日本語")             // Unicode/UTF-8 -> ints
```

(Slices are related to arrays – more later.)

Constants

Numeric constants are "ideal numbers": no size or sign, hence no L or U or UL endings.

```
077  // octal
0xFEEDBEEEEEEEEEEEEEEEEEEEEF  // hexadecimal
1 << 100
```

There are integer and floating-point ideal numbers; syntax of literal determines type:

```
1.234e5   // floating-point
1e2       // floating-point
3.2i      // imaginary floating-point
100       // integer
```

Constant Expressions

Floating point and integer constants can be combined at will, with the type of the resulting expression determined by the type of the constants. The operations themselves also depend on the type.

```
2*3.14  // floating point: 6.28
3./2    // floating point: 1.5
3/2     // integer: 1
3+2i    // complex: 3.0+2.0i

// high precision:
const Ln2 = 0.693147180559945309417232121458176568075
const Log2E = 1/Ln2  // accurate reciprocal
```

Representation is "big enough" (1024 bits now).

Consequences of ideal numbers

The language permits the use of constants without explicit conversion if the value can be represented (there's no conversion necessary; the value is OK):

```
var million int = 1e6  // float syntax OK here
math.Sin(1)
```

Constants must be representable in their type.
Example: `^0` is `-1` which is not in range 0-255.

```
uint8(^0)   // bad: -1 can't be represented
^uint8(0)   // OK
uint8(350)  // bad: 350 can't be represented
uint8(35.0) // OK: 35
uint8(3.5)  // bad: 3.5 can't be represented
```

Declarations

Declarations are introduced by a keyword (`var`, `const`, `type`, `func`) and are reversed compared to C:

```
var i int
const PI = 22./7.
type Point struct { x, y int }
func sum(a, b int) int { return a + b }
```

Why are they reversed? Earlier example:

```
var p, q *int
```

Both `p` and `q` have type `*int`. Also functions read better and are consistent with other declarations. And there's another reason, coming up.

Var

Variable declarations are introduced by var.

They may have a type or an initialization expression; one or both must be present. Initializers must match variables (and types!).

```
var i int
var j = 365.245
var k int = 0
var l, m uint64 = 1, 2
var nanoseconds int64 = 1e9  // float64 constant!
var inter, floater, stringer = 1, 2.0, "hi"
```

Distributing var

Annoying to type `var` all the time. Group with parens:

```
var (
    i int
    j = 356.245
    k int = 0
    l, m uint64 = 1, 2
    nanoseconds int64 = 1e9
    inter, floater, stringer = 1, 2.0, "hi"
)
```

Applies to `const`, `type`, `var` but not `func`.

Google

The := "short declaration"

Within functions (only), declarations of the form

```
var v = value
```

can be shortened to

```
v := value
```

(Another reason for the name/type reversal.)

The type is that of the value (for ideal numbers, get `int` or `float64` or `complex128`, accordingly.)

```
a, b, c, d, e := 1, 2.0, "three", FOUR, 5e0i
```

These are used a lot and are available in places such as `for` loop initializers.

Const

Constant declarations are introduced by `const`.

They must have a "constant expression", evaluated at compile time, as initializer and may have an optional type specifier.

```
const Pi = 22./7.
const AccuratePi float64 = 355./113
const beef, two, parsnip = "meat", 2, "veg"

const (
    Monday, Tuesday, Wednesday = 1, 2, 3
    Thursday, Friday, Saturday = 4, 5, 6
)
```

Iota

Constant declarations can use the counter `iota`, which starts at `0` in each `const` block and increments at each implicit semicolon (end of line).

```
const (
    Monday = iota   // 0
    Tuesday = iota  // 1
)
```

Shorthand: Previous type and expressions repeat.

```
const (
    loc0, bit0 uint32 = iota, 1<<iota  // 0, 1
    loc1, bit1                         // 1, 2
    loc2, bit2                         // 2, 4
)
```

Google

Type

Type declarations are introduced by `type`.

We'll learn more about types later but here are some examples:

```
type Point struct {
    x, y, z float64
    name    string
}
type Operator func(a, b int) int
type SliceOfIntPointers []*int
```

We'll come back to functions a little later.

New

The built-in function new allocates memory.
Syntax is like a function call, with type as argument, similar to C++. Returns a pointer to the allocated object.

```
var p *Point = new(Point)
v := new(int)   // v has type *int
```

Later we'll see how to build slices and such.

There is no delete or free; Go has garbage collection.

Assignment

Assignment is easy and familiar:

```
a = b
```

But multiple assignment works too:

```
x, y, z = f1(), f2(), f3()
a, b = b, a // swap
```

Functions can return multiple values (details later):

```
nbytes, error := Write(buf)
```

Google

Control structures

Similar to C, but different in significant ways.

Go has `if`, `for` and `switch` (plus one more to appear later).

As stated before, no parentheses, but braces mandatory.

They are quite regular when seen as a set. For instance, `if`, `for` and `switch` all accept initialization statements.

Google

Forms of control structures

Details follow but in general:

The `if` and `switch` statements come in 1- and 2-element forms, described below.

The `for` loop has 1- and 3-element forms:

single-element is C's `while`:

```
for a {}
```

triple-element is C's `for`:

```
for a;b;c {}
```

In any of these forms, any element can be empty.

Google

If

Basic form is familiar, but no dangling else problem:

```
if x < 5 { less() }
if x < 5 { less() } else if x == 5 { equal() }
```

Initialization statement allowed; requires semicolon.

```
if v := f(); v < 10 {
    fmt.Printf("%d less than 10\n", v)
} else {
    fmt.Printf("%d not less than 10\n", v)
}
```

Useful with multivariate functions:

```
if n, err = fd.Write(buf); err != nil { ... }
```

Missing condition means `true`, which is not too useful in this context but handy in `for`, `switch`.

Google

For

Basic form is familiar:

```
for i := 0; i < 10; i++ { ... }
```

Missing condition means `true`:

```
for ;; { fmt.Printf("looping forever") }
```

But you can leave out the semis too:

```
for { fmt.Printf("Mine! ") }
```

Don't forget multivariate assigments:

```
for i,j := 0,N; i < j; i,j = i+1,j-1 {...}
```

(There's no comma operator as in C.)

Google

Switch details

Switches are somewhat similar to C's.

But there are important syntactic and semantic differences:

- expressions need not be constant or even `int`.
- no automatic fall through
- instead, lexically last statement can be `fallthrough`
- multiple cases can be comma-separated

```
switch count%7 {
    case 4,5,6: error()
    case 3: a *= v; fallthrough
    case 2: a *= v; fallthrough
    case 1: a *= v; fallthrough
    case 0: return a*v
}
```

Google

Switch

Go's `switch` is more powerful than C's. Familiar form:

```
switch a {
    case 0: fmt.Printf("0")
    default: fmt.Printf("non-zero")
}
```

The expressions can be any type and a missing switch expression means `true`. Result: `if-else` chain:

```
a, b := x[i], y[j]
switch {
    case a < b:  return -1
    case a == b: return 0
    case a > b:  return 1
}
```

or

```
switch a, b := x[i], y[j]; { ... }
```

Google

Break, continue, etc.

The `break` and `continue` statements work as in C.

They may specify a label to affect an outer structure:

```
Loop: for i := 0; i < 10; i++ {
    switch f(i) {
        case 0, 1, 2: break Loop
    }
    g(i)
}
```

Yes Ken, there is a `goto`.

Functions

Functions are introduced by the `func` keyword. Return type, if any, comes after parameters. The `return` does as you expect.

```
func square(f float64) float64 { return f*f }
```

A function can return multiple values. If so, the return types are a parenthesized list.

```
func MySqrt(f float64) (float64, bool) {
    if f >= 0 { return math.Sqrt(f), true }
    return 0, false
}
```

The blank identifier

What if you care only about the first value returned by `MySqrt`? Still need to put the second one somewhere.

Solution: the blank identifier, `_` (underscore). Predeclared, can always be assigned any value, which is discarded.

```
// Don't care about boolean from MySqrt.
val, _ = MySqrt(foo())
```

Handy in other contexts still to be presented.

Google

Functions with result variables

The result "parameters" are actual variables you can use if you name them.

```
func MySqrt(f float64) (v float64, ok bool) {
    if f >= 0 { v,ok = math.Sqrt(f), true }
    else { v,ok = 0,false }
    return v,ok
}
```

The result variables are initialized to "zero" (`0`, `0.0`, `false` etc. according to type; more in a sec).

```
func MySqrt(f float64) (v float64, ok bool) {
    if f >= 0 { v,ok = math.Sqrt(f), true }
    return v,ok
}
```

Google

The empty return

Finally, a `return` with no expressions returns the existing value of the result variables. Two more versions of `MySqrt`:

```
func MySqrt(f float64) (v float64, ok bool) {
    if f >= 0 { v,ok = math.Sqrt(f), true }
    return  // must be explicit
}

func MySqrt(f float64) (v float64, ok bool) {
    if f < 0 { return }  // error case
    return math.Sqrt(f),true
}
```

Google

What was that about zero?

All memory in Go is initialized. All variables are initialized upon execution of their declaration. Without an initializing expression, the "zero value" of the type is used. The loop

```
for i := 0; i < 5; i++ {
    var v int
    fmt.Printf("%d ", v)
    v = 5
}
```

will print `0 0 0 0 0`.

The zero value depends on the type: numeric `0`; boolean `false`; empty string `""`; nil pointer, map, slice, channel; zeroed struct, etc.

Google

Defer

The `defer` statement executes a function (or method) when the enclosing function returns. The arguments are evaluated at the point of the `defer`; the function call happens upon `return`.

```
func data(fileName string) string {
    f := os.Open(fileName)
    defer f.Close()
    contents := io.ReadAll(f)
    return contents
}
```

Useful for closing fds, unlocking mutexes, etc.

Google

One function invocation per defer

Each `defer` that executes queues a function call to execute, in LIFO order, so

```
func f() {
    for i := 0; i < 5; i++ {
        defer fmt.Printf("%d ", i)
    }
}
```

prints `4 3 2 1 0`. You can close all those fds or unlock those mutexes at the end.

Tracing with defer

```
func trace(s string) { fmt.Println("entering:", s) }
func untrace(s string) { fmt.Println("leaving:", s) }

func a() {
    trace("a")
    defer untrace("a")
    fmt.Println("in a")
}
func b() {
    trace("b")
    defer untrace("b")
    fmt.Println("in b")
    a()
}

func main() { b() }
```

But we can do it more neatly...

Args evaluate now, defer later

```
func trace(s string) string {
    fmt.Println("entering:", s)
    return s
}
func un(s string) {
    fmt.Println("leaving:", s)
}
func a() {
    defer un(trace("a"))
    fmt.Println("in a")
}
func b() {
    defer un(trace("b"))
    fmt.Println("in b")
    a()
}
func main() { b() }
```

Google

Function literals

As in C, functions can't be declared inside functions – but function literals can be assigned to variables.

```
func f() {
    for i := 0; i < 10; i++ {
        g := func(i int) { fmt.Printf("%d",i) }
        g(i)
    }
}
```

Function literals are closures

Function literals are indeed closures.

```
func adder() (func(int) int) {
    var x int
    return func(delta int) int {
        x += delta
        return x
    }
}
f := adder()
fmt.Print(f(1))
fmt.Print(f(20))
fmt.Print(f(300))
```

Prints `1 21 321` - accumulating in `f`'s `x`.

Google

Program construction

Google

Packages

A program is constructed as a "package", which may use facilities from other packages.

A Go program is created by linking together a set of packages.

A package may be built from multiple source files.

Names in imported packages are accessed through a "qualified identifier": `packagename.Itemname`.

Source file structure

Every source file contains:

- a package clause (which package it's in); that name is the default name used by importers.

```
package fmt
```

- an optional set of import declarations

```
import "fmt"             // use default name
import myFmt "fmt"  // use the name myFmt
```

- zero or more global or "package-level" declarations.

A single-file package

```
package main // this file is part of package "main"

import "fmt" // this file uses package "fmt"

const hello = "Hello, 世界\n"

func main() {
    fmt.Print(hello)  // fmt is imported pkg's name
}
```

main and main.main

Each Go program contains a package called `main` and its `main` function, after initialization, is where execution starts, analogous with the global `main()` in C, C++.

The `main.main` function takes no arguments and returns no value. The program exits – immediately and successfully – when `main.main` returns.

Google

The os package

Package os provides Exit and access to file I/O, command-line arguments, etc.

```
// A version of echo(1)
package main

import (
    "fmt"
    "os"
)
func main() {
    if len(os.Args) < 2 {  // length of argument slice
        os.Exit(1)
    }
    for i := 1; i < len(os.Args); i++ {
        fmt.Printf("arg %d: %s\n", i, os.Args[i])
    }
}  // falling off end == os.Exit(0)
```

Global and package scope

Within a package, all global variables, functions, types, and constants are visible from all the package's source files.

For clients (importers) of the package, names must be upper case to be visible: global variables, functions, types, constants, plus methods and structure fields for global variables and types.

```
const hello = "you smell"        // package visible
const Hello = "you smell nice"   // globally visible
const _Bye  = "stinko!"          // _ is not upper
```

Very different from C/C++: no `extern`, `static`, `private`, `public`.

Google

Initialization

Two ways to initialize global variables before execution of `main.main`:

1) A global declaration with an initializer
2) Inside an `init()` function, of which there may be any number in each source file.

Package dependency guarantees correct execution order.

Initialization is always single-threaded.

Initialization example

```
package transcendental
import "math"
var Pi float64
func init() {
    Pi = 4*math.Atan(1)  // init function computes Pi
}
====
package main
import (
    "fmt"
    "transcendental"
)
var twoPi = 2*transcendental.Pi  // decl computes twoPi
func main() {
    fmt.Printf("2*Pi = %g\n", twoPi)
}
====
```

Output: `2*Pi = 6.283185307179586`

Package and program construction

To build a program, the packages, and the files within them, must be compiled in the correct order.

Package dependencies determine the order in which to build packages.

Within a package, the source files must all be compiled together. The package is compiled as a unit, and conventionally each directory contains one package. Ignoring tests,

```
cd mypackage
6g *.go
```

Usually we use `make`; Go-specific tool is coming.

Building the fmt package

```
% pwd
/Users/r/go/src/pkg/fmt
% ls
Makefile fmt_test.go format.go print.go # ...
% make  # hand-written but trivial
% ls
Makefile _go_.6 _obj fmt_test.go format.go print.go # ...
% make clean; make
...
```

Objects are placed into the subdirectory `_obj`.

`Makefiles` are written using helpers called `Make.pkg` and so on; see sources.

Testing

To test a package, write a set of Go source files within the same package; give the files names of the form `*_test.go`.

Within those files, global functions with names starting with `Test[^a-z]` will be run by the testing tool, `gotest`. Those functions should have signature

```
func TestXxx(t *testing.T)
```

The `testing` package provides support for logging, benchmarking, error reporting.

An example test

Interesting pieces from `fmt_test.go`:

```
package fmt  // package is fmt, not main
import (
    "testing"
)
func TestFlagParser(t *testing.T) {
    var flagprinter flagPrinter
    for i := 0; i < len(flagtests); i++ {
        tt := flagtests[i]
        s := Sprintf(tt.in, &flagprinter)
        if s != tt.out {
            // method call coming up - obvious syntax.
            t.Errorf("Sprintf(%q, &flagprinter) => %q,"
                    + " want %q", tt.in, s, tt.out)
        }
    }
}
```

Testing: gotest

```
% ls
Makefile fmt.a fmt_test.go format.go print.go # ...
% gotest # by default, does all *_test.go
PASS
wally=% gotest -v fmt_test.go
=== RUN  fmt.TestFlagParser
--- PASS: fmt.TestFlagParser (0.00 seconds)
=== RUN  fmt.TestArrayPrinter
--- PASS: fmt.TestArrayPrinter (0.00 seconds)
=== RUN  fmt.TestFmtInterface
--- PASS: fmt.TestFmtInterface (0.00 seconds)
=== RUN  fmt.TestStructPrinter
--- PASS: fmt.TestStructPrinter (0.00 seconds)
=== RUN  fmt.TestSprintf
--- PASS: fmt.TestSprintf (0.00 seconds) # plus lots more
PASS
%
```

An example benchmark

Benchmarks have signature

```
func BenchmarkXxxx(b *testing.B)
```

and loop over `b.N`; testing package does the rest.
Here is a benchmark example from `fmt_test.go`:

```
package fmt  // package is fmt, not main
import (
    "testing"
)

func BenchmarkSprintfInt(b *testing.B) {
    for i := 0; i < b.N; i++ {
        Sprintf("%d", 5)
    }
}
```

Google

Benchmarking: gotest

```
% gotest -bench="."  # regular expression identifies which
fmt_test.BenchmarkSprintfEmpty    5000000       310 ns/op
fmt_test.BenchmarkSprintfString   2000000       774 ns/op
fmt_test.BenchmarkSprintfInt      5000000       663 ns/op
fmt_test.BenchmarkSprintfIntInt   2000000       969 ns/op
...
. . .
%
```

Libraries

Libraries are just packages.
The set of libraries is modest but growing.
Some examples:

Package	Purpose	Examples
`fmt`	formatted I/O	`Printf, Scanf`
`os`	OS interface	`Open, Read, Write`
`strconv`	numbers <-> strings	`Atoi, Atof, Itoa`
`io`	generic I/O	`Copy, Pipe`
`flag`	flags: `--help` etc.	`Bool, String`
`log`	event logging	`Logger, Printf`
`regexp`	regular expressions	`Compile, Match`
`template`	HTML, etc.	`Parse, Execute`
`bytes`	byte arrays	`Compare, Buffer`

Google

A little more about fmt

The `fmt` package contains familiar names in initial caps:

`Printf` – print to standard output
`Sprintf` – returns a string
`Fprintf` – writes to `os.Stderr` etc. (tomorrow)

but also

`Print`, `Sprint`, `Fprint` – no format
`Println`, `Sprintln`, `Fprintln` – no format, adds spaces, final `\n`

```
fmt.Printf("%d %d %g\n", 1, 2, 3.5)
fmt.Print(1, " ", 2, " ", 3.5, "\n")
fmt.Println(1, 2, 3.5)
```

Each produces the same result: `"1 2 3.5\n"`

Google

Library documentation

Source code contains comments.
Command line or web tool pulls them out.
Link:

http://golang.org/pkg/

Command:

```
% godoc fmt
% godoc fmt Printf
```

Exercise

Google

Exercise: Day 1

Set up a Go environment - see
http://golang.org/doc/install.html

You all know what the Fibonacci series is. Write a package to implement it. There should be a function to get the next value. (You don't have structs yet; can you find a way to save state without globals?) But instead of addition, make the operation settable by a function provided by the user. Integers? Floats? Strings? Up to you.

Write a `gotest` test for your package.

Next lesson

Composite types

Methods

Interfaces

The
Go
Programming Language

Part 1

Rob Pike
r@google.com
(updated June 2011)