Distributed Systems

05r. Go Programming

Paul Krzyzanowski
TA: Yuanzhen Gu
Rutgers University
Fall 2014
Motivation

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 are poorly supported.
- Multi-core seen as crisis not opportunity.
Go’s goal

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
package main
import "fmt"
func main() {
    fmt.Print("Hello, World \n")
}

Syntax overview

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

```plaintext
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:

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

Note: no parentheses, but braces required.
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 rules!
- Thus, no semis needed in this program:

```go
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.*
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.
Declarations

Declarations are introduced by a keyword (\texttt{var, const, type, func}) and are reversed compared to C:

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

Why are they reversed? Earlier example:

\begin{verbatim}
var p, q *int
\end{verbatim}

Both \texttt{p} and \texttt{q} have type \texttt{*int}

Also functions read better and are consistent with other declarations. And there's another reason, coming up.
• Variable declarations are introduced by \texttt{var}

• They may have a type or an initialization expression
  – One or both must be present

• Initializers must match variables (and types!)

\begin{verbatim}
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"
\end{verbatim}
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.
Constant declarations are introduced by const

They must have a "constant expression", evaluated at compile time, as an 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  
    )
```
Type

Type declarations are introduced by type.

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

```go
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.

```go
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 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)```
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.
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
for

Basic form is familiar:

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

Missing condition means true:

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

But you can leave out the semis too:

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

Don't forget multivariate assignments:

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

(There's no comma operator as in C)
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

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

```go
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:

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

or

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

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

They may specify a label to affect an outer structure:

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

Yes, 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.

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

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

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

```go
define data(fileName string) string {
    f := os.Open(fileName)
deferr f.Close()
    contents := io.ReadAll(f)
    retrun contents
}
```

• Useful for closing fds, unlocking mutexes, etc.
Program construction - 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`.
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
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
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.
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.
Arrays

Arrays are values, not implicit pointers as in C. You can take an array's address, yielding a pointer to the array (for instance, to pass it efficiently to a function):

```go
func f(a [3]int) { fmt.Println(a) }
func fp(a *[3]int) { fmt.Println(a) }

func main() {
    var ar [3] int
    f(ar) // passes a copy of ar
    fp(&ar) // passes a pointer to ar
}
```

Output (Print and friends know about arrays):

```
[0 0 0]
&[0 0 0]
```
Pointers to array literals

You can take the address of an array literal to get a pointer to a newly created instance:

```go
func fp(a *[3]int) { fmt.Println(a) }

func main() {
    for i := 0; i < 3; i++ {
        fp(&[3]int{i, i*i, i*i*i})
    }
}
```

Output:

```
&[0 0 0]
&[1 1 1]
&[2 4 8]
```
Maps

Maps are another reference type. They are declared like this:

```go
var m map[string]float64
```

This declares a map indexed with key type `string` and value type `float64`.

It is analogous to the C++ type `*map<string,float64>` (note the `*`).

Given a map `m`, `len(m)` returns the number of keys.
Map creation

• As with a slice, a map variable refers to nothing; you must put something in it before it can be used.

• Three ways:

1. **Literal**: list of colon-separated key:value pairs
   
   ```go
   m = map[string]float64{"1":1, "pi":3.1415}
   ```

2. **Creation**
   
   ```go
   m = make(map[string]float64) // make not new
   ```

3. **Assignment**
   
   ```go
   var m1 map[string]float64
   m1 = m // m1 and m now refer to same map
   ```
Indexing a map

(Next few examples all use

\[ m = \text{map}[\text{string}]\text{float64}\{"1":1, "pi":3.1415\} \]

)  

Access an element as a value; if not present, get zero value for the map's value type:

\[
\text{one} := m["1"]
\]

\[
\text{zero} := m["not present"] // Sets zero to 0.0.
\]

Set an element (setting twice updates value for key)

\[
m["2"] = 2
\]

\[
m["2"] = 3 // mess with their heads
\]
Testing existence

To test if a key is present in the map, we can use a multi-value assignment, the "comma ok" form:

```go
m := map[string]float64{"1":1, "pi":3.1415}
var value float64
var present bool
value, present = m[x]
```

or idiomatically

```go
value, ok := m[x] // hence, the "comma ok" form
```

If `x` is present in the map, sets the boolean to `true` and the value to the entry for the key. If not, sets the boolean to `false` and the value to the zero for its type.
Deleting an entry in the map is a multi-variate assignment to the map entry:

```go
m = map[string]float64{"1":1.0, "pi":3.1415}
var keep bool
var value float64
var x string = f()
m[x] = v, keep
```

If `keep` is `true`, assigns `v` to the map; if `keep` is `false`, deletes the entry for key `x`. So to delete an entry:

```go
m[x] = 0, false // deletes entry for x
```
Structs

Structs should feel very familiar: simple declarations of data fields.

```go
var p struct {
    x, y float64
}
```

More usual:

```go
type Point struct {
    x, y float64
}
var p Point
```

Structs allow the programmer to define the layout of memory
Anonymous fields

• Inside a struct, you can declare fields, such as another struct, without giving a name for the field.

• These are called anonymous fields and they act as if the inner struct is simply inserted or "embedded" into the outer.

• This simple mechanism provides a way to derive some or all of your implementation from another type or types.

• An example follows.
An anonymous struct field

type A struct {
    ax, ay int
}
type B struct {
    A
    bx, by float64
}

B acts as if it has four fields, ax, ay, bx, and by
It’s almost as if B is {ax, ay int; bx, by float64}.
However, literals for B must be filled out in detail:

    b := B{A{1, 2}, 3.0, 4.0}
    fmt.Println(b.ax, b.ay, b.bx, b.by)

Prints 1 2 3 4
Methods on structs

• Go has no classes, but you can attach methods to any type. Yes, (almost) any type.

• The methods are declared, separate from the type declaration, as functions with an explicit receiver

• The obvious struct case:

```go
type Point struct { x, y float64 }
// A method on *Point
func (p *Point) Abs() float64 {
    return math.Sqrt(p.x*p.x + p.y*p.y)
}
```

• Note: explicit receiver (no automatic this), in this case of type *Point, used within the method.
Methods on struct values

A method does not require a pointer as a receiver.

```go
type Point3 struct { x, y, z float64 }

// A method on Point3
func (p Point3) Abs() float64 {
    return math.Sqrt(p.x*p.x + p.y*p.y + p.z*p.z)
}
```

This is a bit expensive, because the Point3 will always be passed to the method by value, but it is valid Go.
Invoking a method

Just as you expect.

```go
p := &Point{ 3, 4 }
fmt.Println(p.Abs()) // will print 5
```

A non-struct example:

```go
type IntVector []int
func (v IntVector) Sum() (s int) {
    for _, x := range v { // blank identifier!
        s += x
    }
    return
}
fmt.Println(IntVector{1, 2, 3}.Sum())
```
• So far, all the types we have examined have been concrete: they implement something

• There is one more type to consider: the interface type
  – It is completely abstract; it implements nothing
  – Instead, it specifies a set of properties an implementation must provide.

• Interface as a concept is very close to that of Java, and Java has an interface type, but the "interface value" concept of Go is novel.
Definition of an interface

• The word "interface" is a bit overloaded in Go: there is the concept of an interface, and there is an interface type, and then there are values of that type. First, the concept.

• **Definition:** An interface is a set of methods.

• To turn it around, the methods implemented by a concrete type such as a `struct` form the interface of that type.
An example

type MyFloat float64
func (f MyFloat) Abs() float64 {
    if f < 0 { return float64(-f) }
    return f
}

MyFloat implements AbsInterface even though float64 does not

(Aside: MyFloat is not a "boxing" of float64; its representation is identical to float64.)
Comparison

• In C++ terms, an interface type is like a pure abstract class, specifying the methods but implementing none of them

• In Java terms, an interface type is much like a Java interface

• However, in Go there is a major difference:
  – A type does not need to declare the interfaces it implements, nor does it need to inherit from an interface type
  – If it has the methods, it implements the interface.

• Some other differences will become apparent
Goroutines

Terminology:

– There are many terms for "things that run concurrently": processes, threads, coroutines, POSIX threads, NPTL threads, lightweight processes, ..., but

– These all mean slightly different things. None mean exactly how Go does concurrency

– We introduce a new term: goroutine
A **goroutine** is a Go function or method executing **concurrently in the same address space** as other goroutines.

- A running program consists of one or more goroutines.

It's not the same as a thread, coroutine, process, etc. It's a **goroutine**.

**Note:** **Concurrency** and **parallelism** are different concepts

- Look them up if you don't understand the difference.

There are many concurrency questions. They will be addressed later; for now, just assume it all works as advertised.
Starting a goroutine

Invoke a function or method and say *go*:

```go
func IsReady(what string, minutes int64) {
    time.Sleep(minutes * 60*1e9) // Unit is nanosecs.
    fmt.Println(what, "is ready")
}

go IsReady("tea", 6)
go IsReady("coffee", 2)
fmt.Println("I'm waiting...")
```

Prints:

```
I'm waiting... (right away)
coffee is ready (2 minutes later)
tea is ready (6 minutes later)
```
Channels in Go

• Unless two goroutines can communicate, they can't coordinate

• Go has a type called a channel that provides communication and synchronization capabilities

• It also has special control structures that build on channels to make concurrent programming easy
The Channel Type

In its simplest form the type looks like this:

```
chan elementType
```

With a value of this type, you can send and receive items of `elementType`.

Channels are a reference type, which means if you assign one `chan` variable to another, both variables access the same channel. It also means you use `make` to allocate one:

```
var c = make(chan int)
```
The communication operator: `<-`

The arrow points in the direction of data flow.

As a binary operator, `<-` sends the value on the right to the channel on the left:

```go
c := make(chan int)
c <- 1 // send 1 on c (flowing into c)
```

As a prefix unary operator, `<-` receives from a channel:

```go
v = <-c // receive value from c, assign to v
<-c // receive value, throw it away
i := <-c // receive value, initialize i
```
func pump(ch chan int) {
    for i := 0; ; i++ { ch <- i }
}

ch1 := make(chan int)
go pump(ch1) // pump hangs; we run
fmt.Println(<-ch1) // prints 0

Now we start a looping receiver.

func suck(ch chan int) {
    for { fmt.Println(<-ch) }
}

go suck(ch1) // tons of numbers appear

You can still sneak in and grab a value:

fmt.Println(<-ch1) // Prints 314159
In the previous example, pump was like a generator spewing out values. But there was a lot of fuss allocating channels etc. Let's package it up into a function returning the channel of values.

```go
func pump() chan int {
    ch := make(chan int)
    go func() {
        for i := 0; ; i++ { ch <- i }
    }()
    return ch
}

stream := pump()
fmt.Println(<-stream) // prints 0
```

"Function returning channel" is an important idiom
Close

• Key points:
  – Only the sender should call close
  – Only the receiver can ask if channel has been closed
  – Can only ask while getting a value (avoids races)

• Call close only when it's necessary to signal to the receiver that no more values will arrive

• Most of the time, close isn't needed
  – It's not analogous to closing a file

• Channels are garbage-collected regardless
Channel directionality

• In its simplest form a channel variable is an unbuffered (synchronous) value that can be used to send and receive

• A channel type may be annotated to specify that it may only send or only receive:

```go
var recvOnly <-chan int
var sendOnly chan<- int
```
Channel directionality (II)

• All channels are created bidirectional, but we can assign them to directional channel variables

• Useful for instance in functions, for (type) safety:

```go
func sink(ch <-chan int) {
    for { <-ch }
}

func source(ch chan<- int) {
    for { ch <- 1 }
}

c := make(chan int) // bidirectional

go source(c)
go sink(c)
```
Synchronous channels

Synchronous channels are unbuffered. Sends do not complete until a receiver has accepted the value.

c := make(chan int)
go func() {
    time.Sleep(60*1e9)
    x := <-c
    fmt.Println("received", x)
}()
fmt.Println("sending", 10)
c <- 10
fmt.Println("sent", 10)

Output:
sending 10 (happens immediately)
sent 10 (60s later, these 2 lines appear)
received 10
Asynchronous channels

A buffered, asynchronous channel is created by telling `make` the number of elements in the buffer:

```go
c := make(chan int, 50)
go func() {
    time.Sleep(60*1e9)
    x := <-c
    fmt.Println("received", x)
}()
fmt.Println("sending", 10)
c <- 10
fmt.Println("sent", 10)
```

Output:

- `sending 10` (happens immediately)
- `sent 10` (now)
- `received 10` (60s later)
Resources

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

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

An online book:
The end